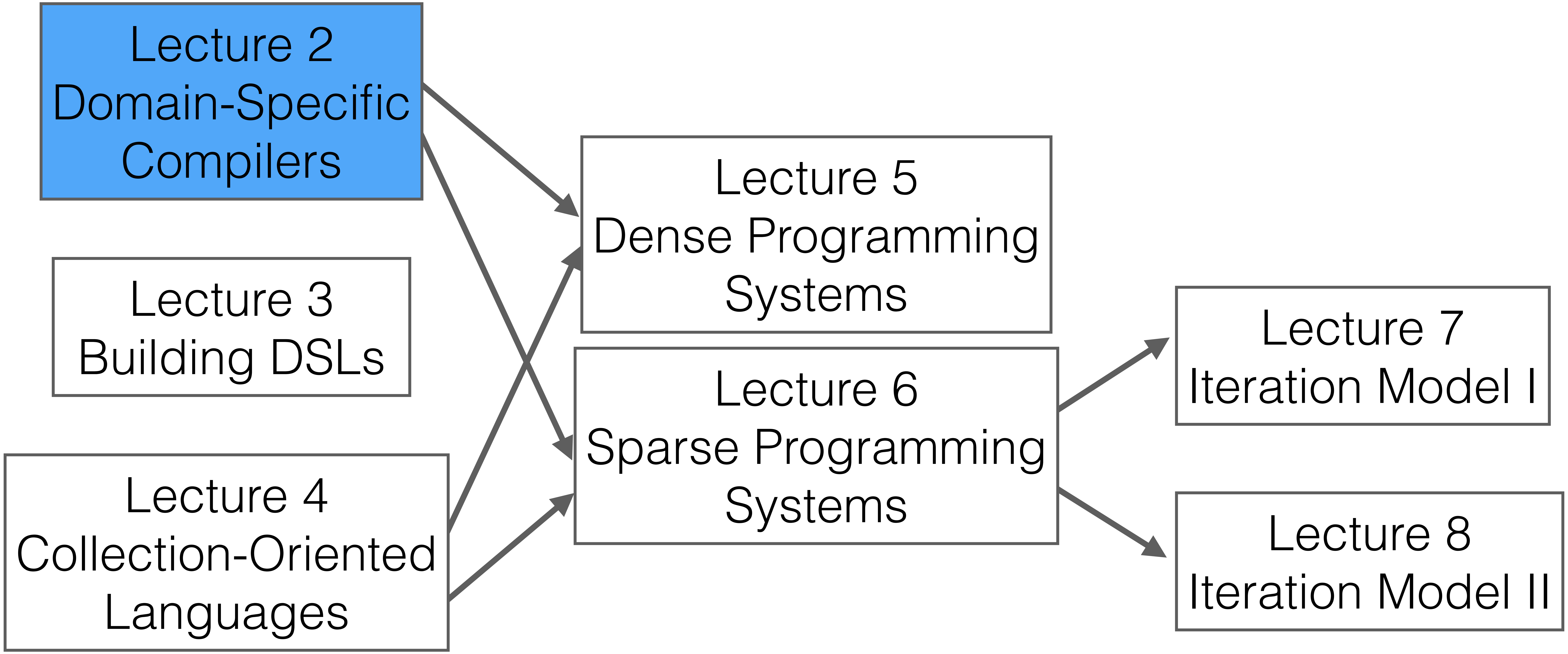
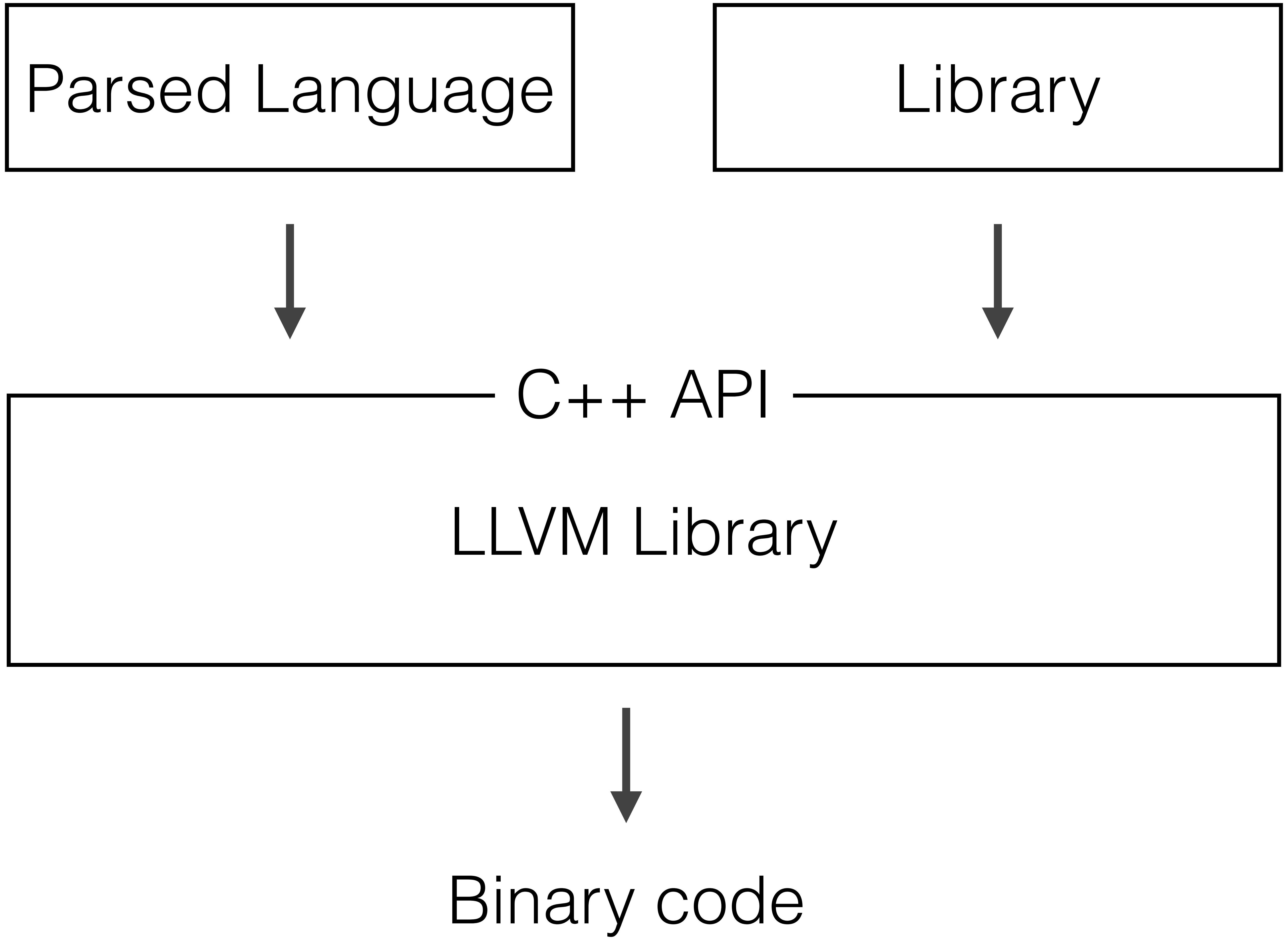
# Lecture 2 — Why Domain-Specific Compilers


Stanford CS343D (Winter 2026)
Fred Kjolstad


Lecture 2
Domain-Specific
Compilers
Lecture 3
Building DSLs
Lecture 4
Collection-Oriented
Languages
Lecture 5
Dense Programming
Systems
Lecture 6
Sparse Programming
Systems
Lecture 7
Iteration Model I
Lecture 8
Iteration Model II

# Languages vs libraries: LLVM is a compiler for general languages, yet it is also just a C++ library with no parser




```
// a*b
Value* mul = builder.CreateMul(a, b);
```

https://llvm.org

# Selective highlights from the history of compilers

1950s - Fortran Compiler

- The History of Fortran (Backus 1982)

# Selective highlights from the history of compilers

1950s - Fortran Compiler

- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations

- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

# Selective highlights from the history of compilers

1950s - Fortran Compiler

- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations

- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

1970s - Loop transformations

- Strip-mining and tiling
- Simple loop parallelization
- Dependence testing
- Lots of work at Illinois

# Selective highlights from the history of compilers

1950s - Fortran Compiler

- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations

- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

1970s - Loop transformations

- Strip-mining and tiling
- Simple loop parallelization
- Dependence testing
- Lots of work at Illinois

1980s - Loop analysis and parallelization

- Auto-vectorization
- Auto-parallelization

# Selective highlights from the history of compilers

1950s - Fortran Compiler

- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations

- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

1970s - Loop transformations

- Strip-mining and tiling
- Simple loop parallelization
- Dependence testing
- Lots of work at Illinois

1980s - Loop analysis and parallelization

- Auto-vectorization
- Auto-parallelization

1990s - Polyhedral model

- Polyhedral scheduling
- Polyhedra scanning (code generation)
- Remove cost of object-orientation

# Selective highlights from the history of compilers

1950s - Fortran Compiler
- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations
- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

1970s - Loop transformations
- Strip-mining and tiling
- Simple loop parallelization
- Dependence testing
- Lots of work at Illinois

1980s - Loop analysis and parallelization
- Auto-vectorization
- Auto-parallelization

1990s - Polyhedral model
- Polyhedral scheduling
- Polyhedra scanning (code generation)
- Remove cost of object-orientation

2000s - SSA and LLVM
- SSA becomes widely used
- LLVM makes compilers accessible
- Remove cost of dynamic languages

# Selective highlights from the history of compilers

1950s - Fortran Compiler
- The History of Fortran (Backus 1982)

1960s - Simple expression optimizations
- Common subexpression elimination
- Operator strength reduction
- Program optimization (Allen 1969)

1970s - Loop transformations
- Strip-mining and tiling
- Simple loop parallelization
- Dependence testing
- Lots of work at Illinois

1980s - Loop analysis and parallelization
- Auto-vectorization
- Auto-parallelization

1990s - Polyhedral model
- Polyhedral scheduling
- Polyhedra scanning (code generation)
- Remove cost of object-orientation

2000s - SSA and LLVM
- SSA becomes widely used
- LLVM makes compilers accessible
- Remove cost of dynamic languages

2010s - DSLs and Program Synthesis
- Halide, TensorFlow/XLA, PyTorch, Taco
- Code generation for SQL

# Automatic programming

The compiler as an optimizer

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int j = 0; j < N; j++) {
    int pB2 = i*N + j;
    t += B[pB2] * c[j];
  }
  a[i] = t;
}
```

optimize →

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    t += B[p] * c[j];
  }
  a[i] = t;
}
```

# Automatic programming

## The compiler as an optimizer

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int j = 0; j < N; j++) {
    int pB2 = i*N + j;
    t += B[pB2] * c[j];
  }
  a[i] = t;
}
```

optimize →

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    t += B[p] * c[j];
  }
  a[i] = t;
}
```

## The compiler as generator

$a = Bc$

$\downarrow$ lower

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    t += B[p] * c[j];
  }
  a[i] = t;
}
```

# Automatic programming

## The compiler as an optimizer

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int j = 0; j < N; j++) {
    int pB2 = i*N + j;
    t += B[pB2] * c[j];
  }
  a[i] = t;
}
```

$\xrightarrow{\text{optimize}}$

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    t += B[p] * c[j];
  }
  a[i] = t;
}
```

## The compiler as generator

$$a = Bc$$

$\downarrow$ lower

```
for (int i = 0; i < M; i++) {
  double t = 0.0;
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    t += B[p] * c[j];
  }
  a[i] = t;
}
```

"In short, automatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer. Research in automatic programming is simply research in the implementation of higher-level programming languages."
- David Parnas

# Granularity of generated code

$$A = B \odot (CD)$$

$\big\downarrow$ select

```
Matrix T = gemm(C,D);
Matrix A = spelmul(B,T);
```

Kernel Library

`gemm` `spelmul` `ttv` `spmv` `mttkrp` `ttm` `matadd` `...`

# Granularity of generated code

$$A = B \odot (CD)$$

$\downarrow$ select

```
Matrix T = gemm(C,D);
Matrix A = spelmul(B,T);
```

**Kernel Library**

gemm, ttv, mttkrp, matadd, spelmul, spmv, ttm, ...

$$A = B \odot (CD)$$

$\downarrow$ compile

```
int pA2 = 0;
for (int pB1 = B1_pos[0];
     pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];
  for (int pB2 = B2_pos[pB1];
       pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_crd[pB2];
    double t = 0.0;
    for (int k = 0; k < O; k++) {
      int pC2 = i * O + k;
      int pD2 = k * N + j;
      t += C[pC2] * D[pD2];
    }
    A[pA2++] = B[pB2] * t;
  }
}
```

# What does a compiler do for you?

1. Lets you program a different machine than the one you actually have
   - A high-level language is an imaginary/abstract/virtual machine
   - The compiler automatically programs the actual machine for you
2. Lets you know if you are using the language incorrectly
3. Optimizes the performance of your program

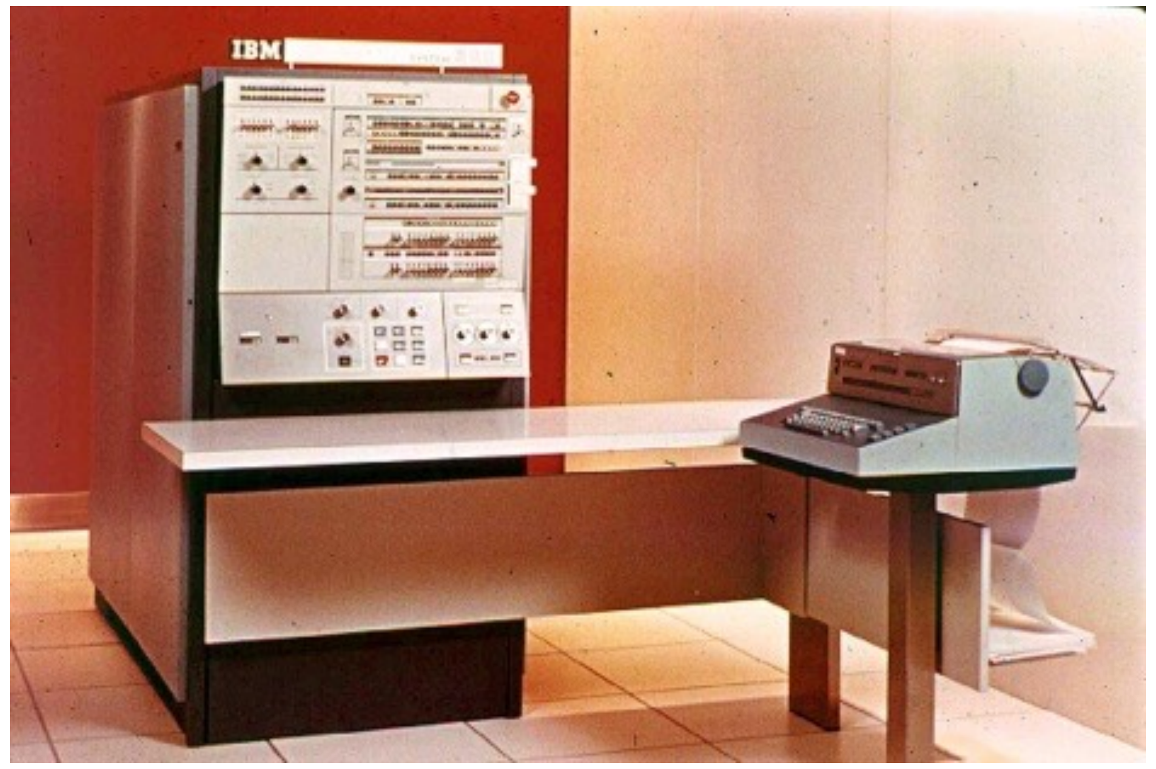
# In the Golden Era of Computing, Performance Engineering Ruled the World

Every programmer was a Performance Engineer

# In the Golden Era of Computing, Performance Engineering Ruled the World

## Every programmer was a Performance Engineer

IBM System/360

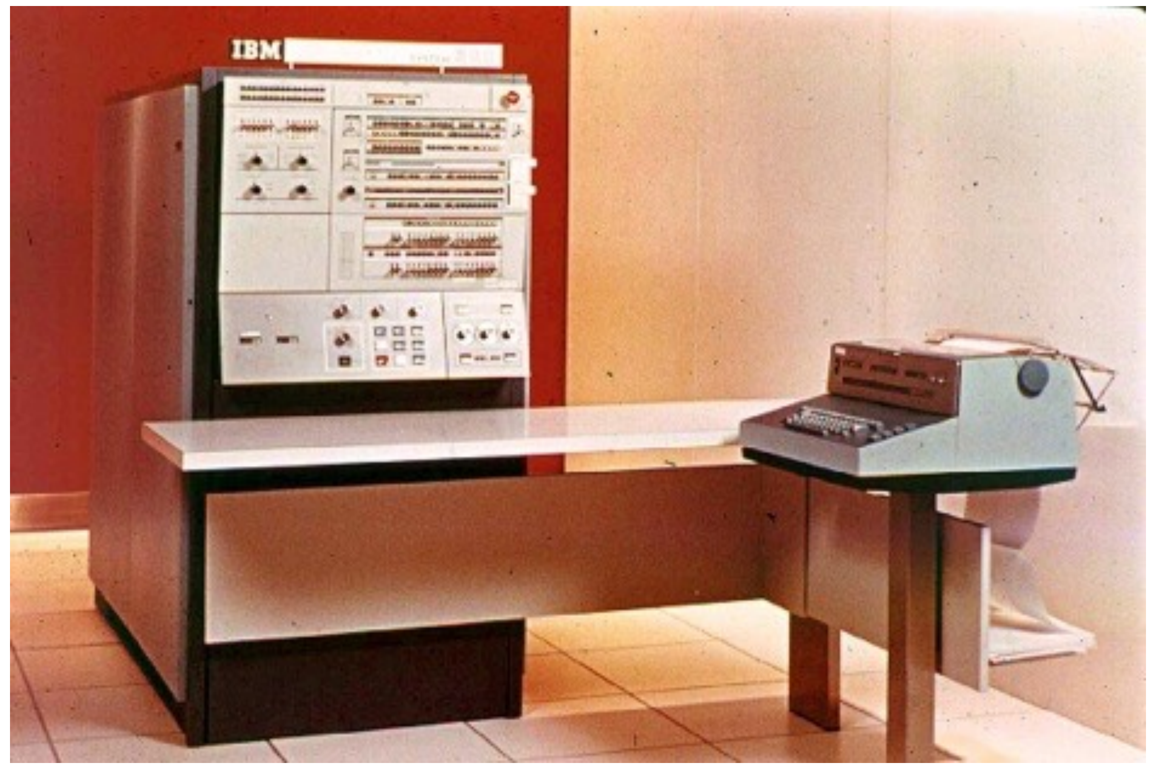



Launched: 1964
Clock rate: 33 KHz
Data path: 32bits
Memory: 524 Kbytes
Cost: $5,000 per month

# In the Golden Era of Computing, Performance Engineering Ruled the World

## Every programmer was a Performance Engineer

IBM System/360




Launched: 1964
Clock rate: 33 KHz
Data path: 32bits
Memory: 524 Kbytes
Cost: $5,000 per month

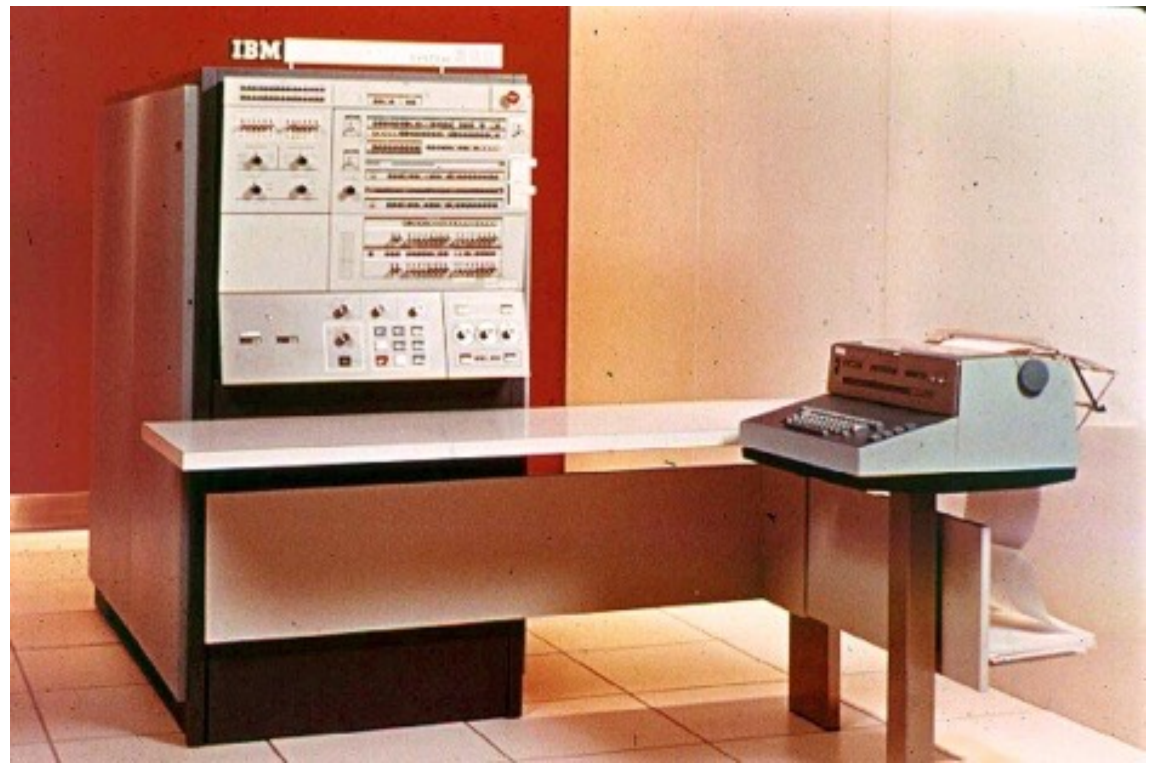
Apple II

Launched: 1977
Clock rate: 1 MHz
Data path: 8 bits
Memory: 48 Kbytes
Cost: $1,395

# In the Golden Era of Computing,
# Performance Engineering Ruled the World

## Every programmer was a Performance Engineer

IBM System/360




| | |
|---|---|
| Launched: | 1964 |
| Clock rate: | 33 KHz |
| Data path: | 32bits |
| Memory: | 524 Kbytes |
| Cost: | $5,000 per month |

Apple II

| | |
|---|---|
| Launched: | 1977 |
| Clock rate: | 1 MHz |
| Data path: | 8 bits |
| Memory: | 48 Kbytes |
| Cost: | $1,395 |

Any useful program would stretch the machine resources
- Program had to be planned around the machine
- Many would not ‘fit’ without intense performance hacks

# Software Properties

What do programmers want to add?

# Software Properties

What do programmers want to add?

- Functionality

# Software Properties

What do programmers want to add?

- Functionality

… and…

# Software Properties

What do programmers want to add?

- Functionality

… and…

- Scalability
- Compatibility
- Correctness
- Clarity
- Low Power
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Testability
- Usability

… and more.

# Software Properties

What do programmers want to add?

- Functionality

… and…

- Scalability
- Compatibility
- Correctness
- Clarity
- Low Power
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Testability
- Usability

… and more.

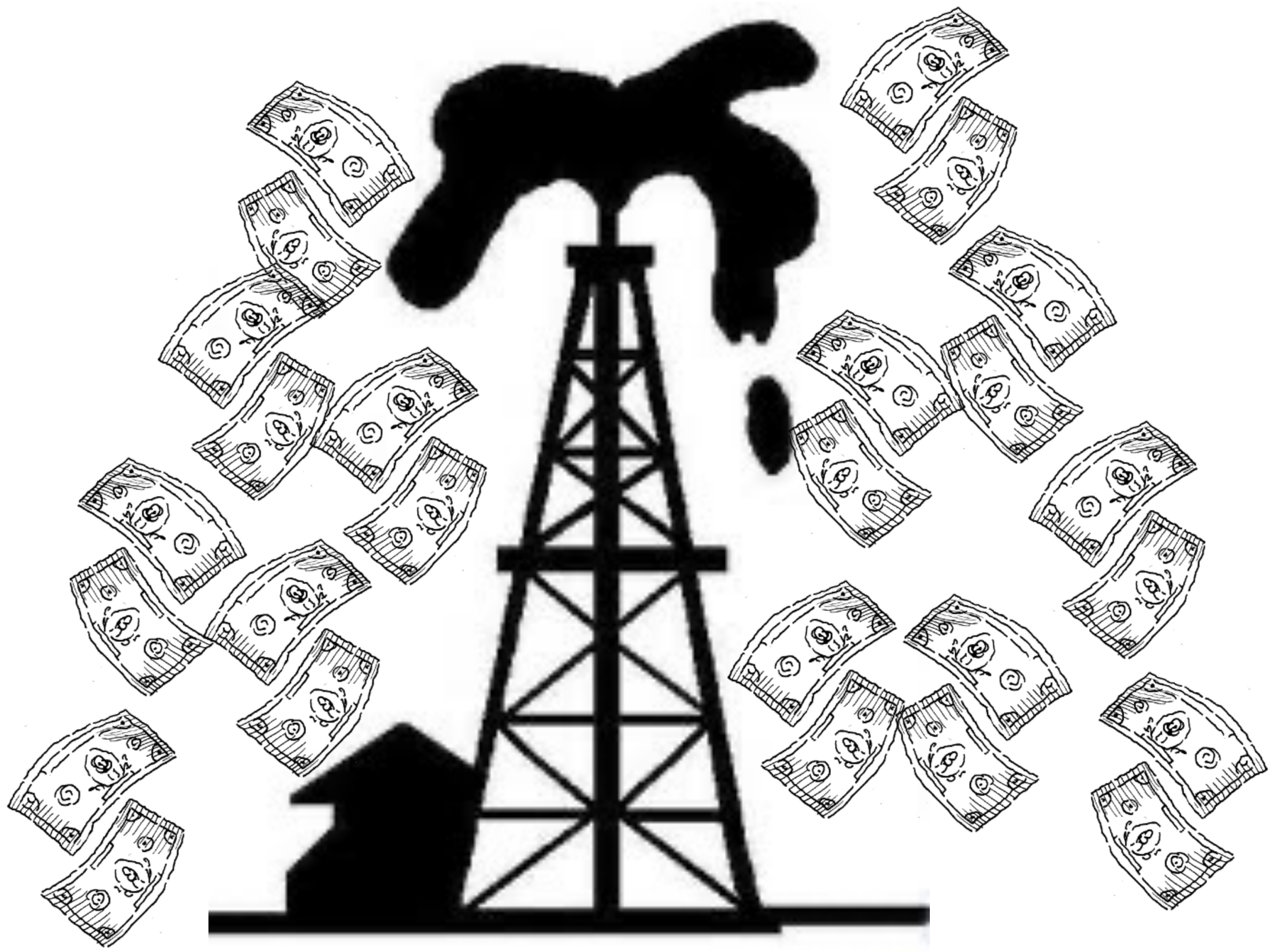
Performance is the currency of computing. You can often "buy" needed properties with performance.

# In the Dominant Era of Computing, Performance became Free

The currency was free

Only need to wait a few months

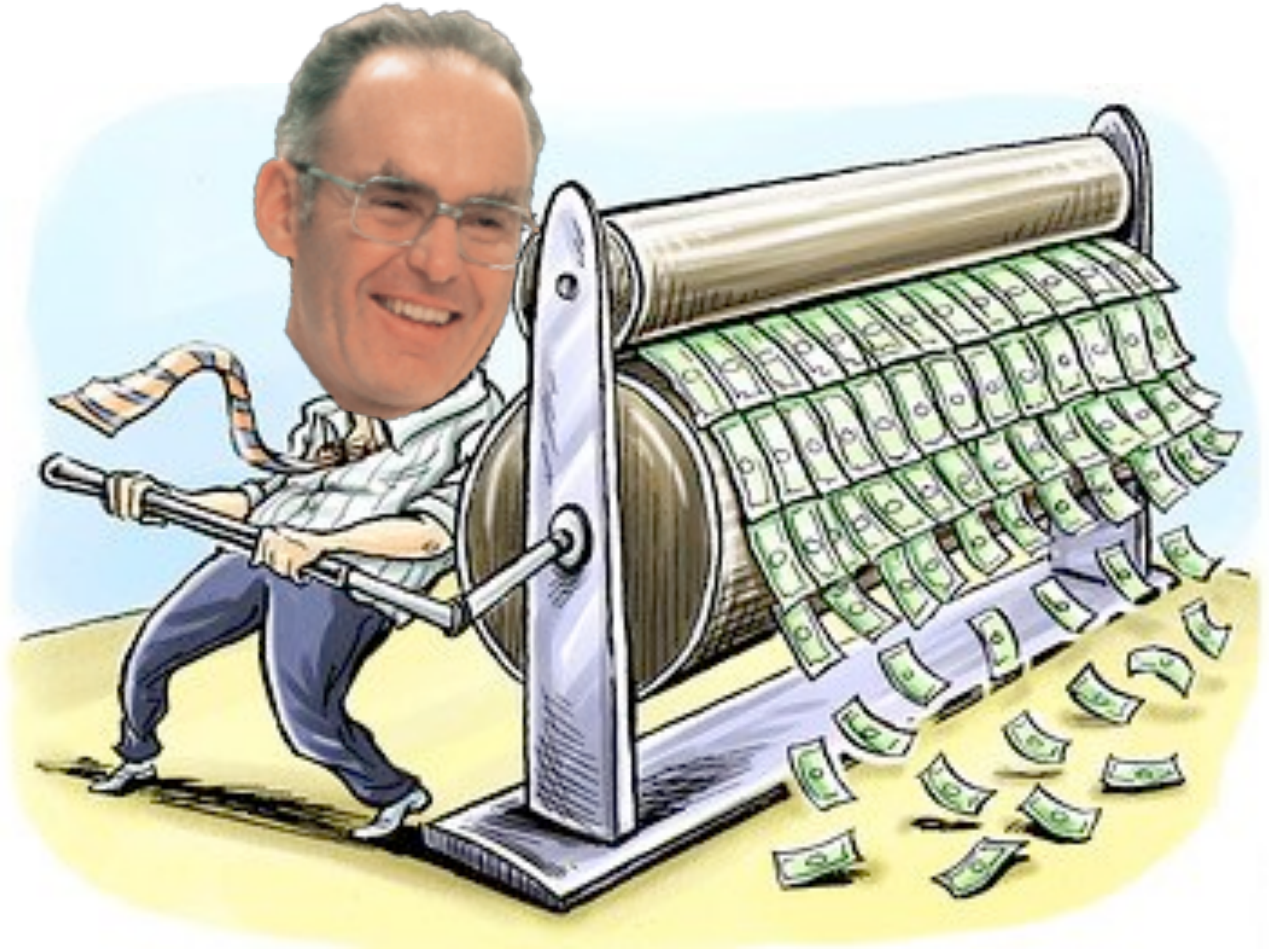
Performance doubled every 2 years



Performance is the currency of computing. You can often "buy" needed properties with performance.

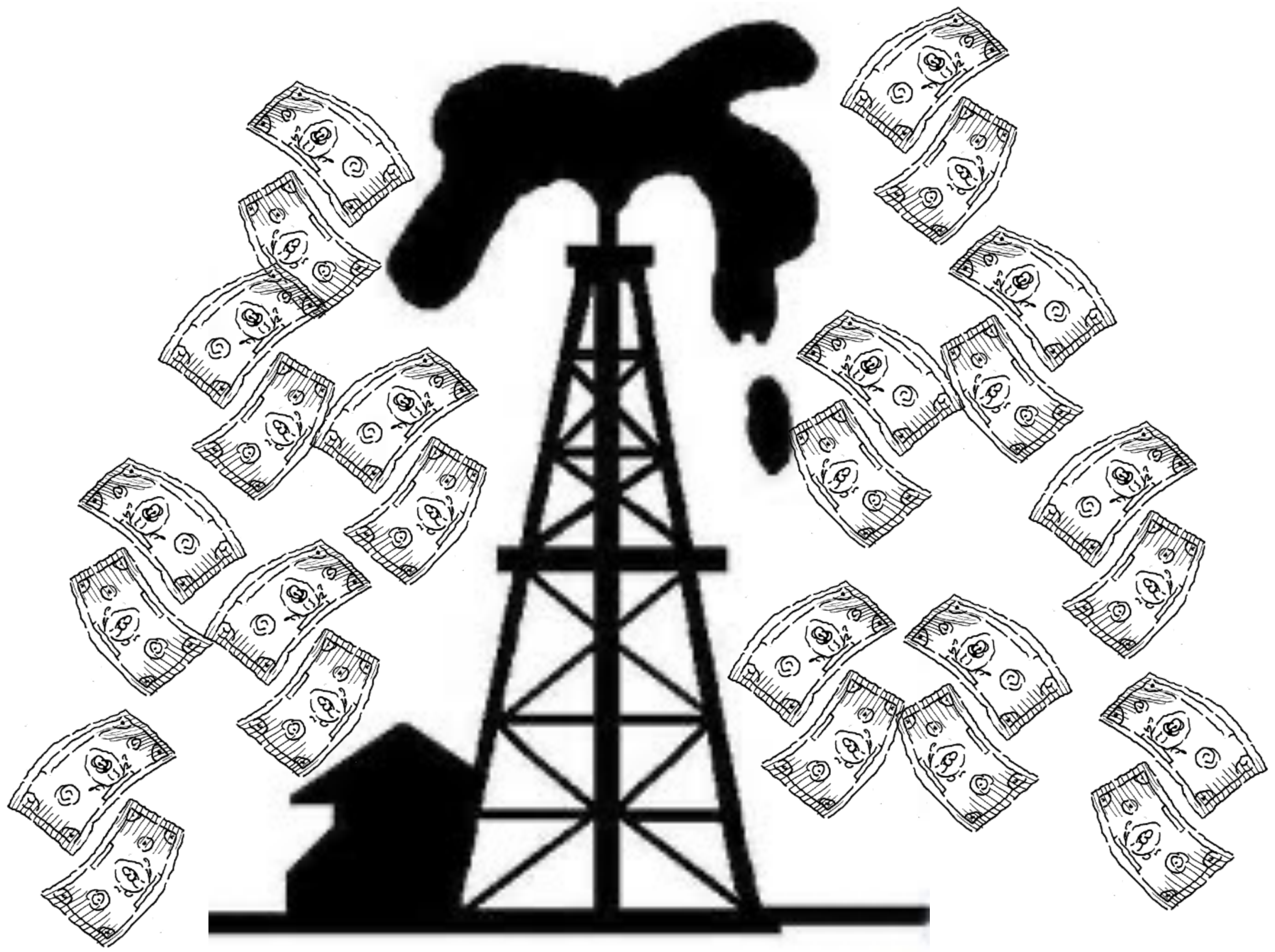
# In the Dominant Era, Performance was Free

Moore's Law and the scaling of clock frequency
= printing press for the currency of performance



# In the Dominant Era, Performance was Free

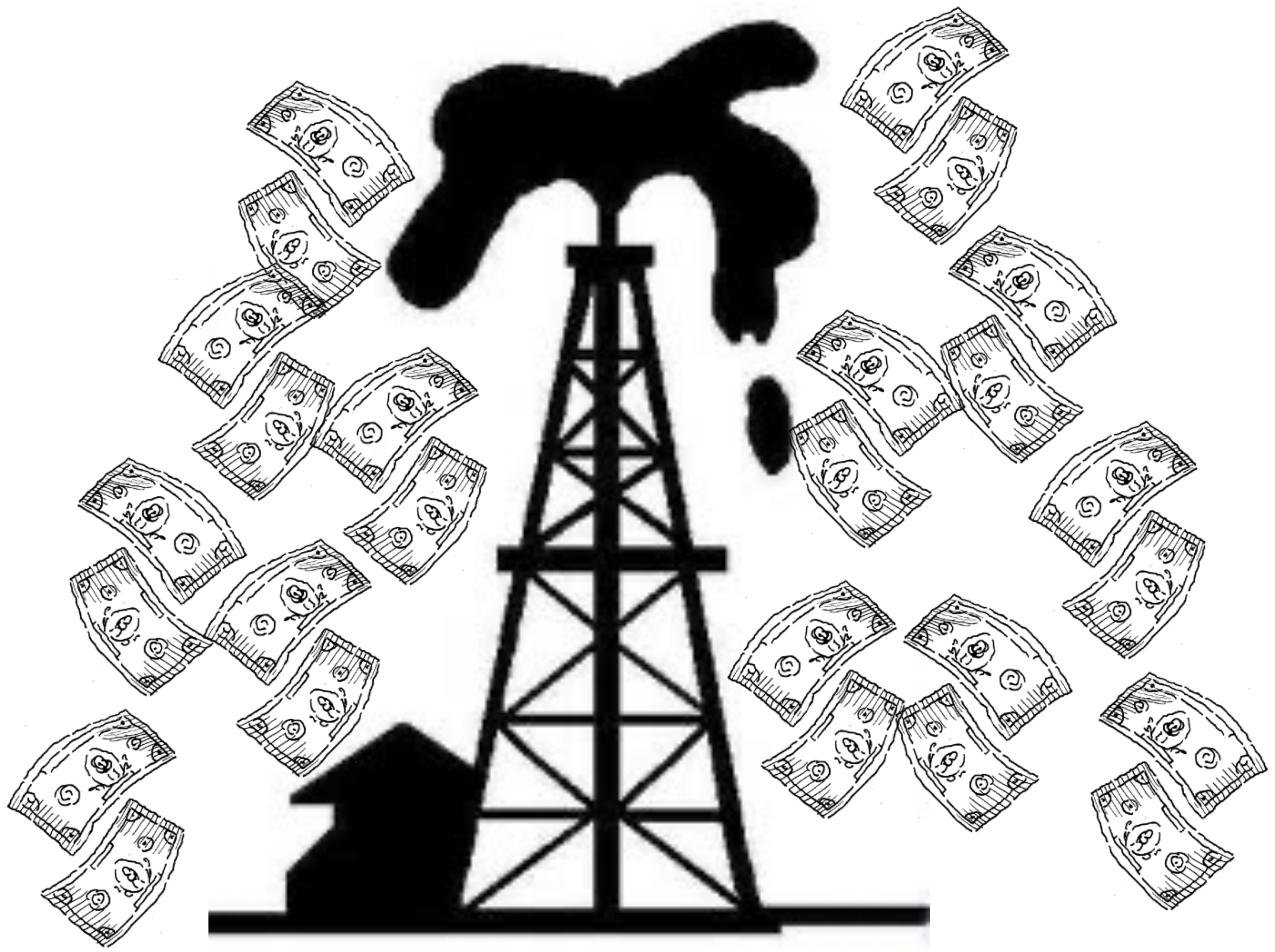
Performance engineering was
'optional' at best and
'irrelevant' for most programmers



Performance is the currency of computing. You can often "buy" needed properties with performance.

# In the Dominant Era, Performance was Free

Performance engineering was
'optional' at best and
'irrelevant' for most programmers



premature optimization is the root of all evil
— Don Knuth

Performance is the currency of computing. You can often "buy" needed properties with performance.

# The Age of Free Performance is Over

Moore's law is not giving free performance any more

Performance is the currency of computing.  You can often "buy" needed properties with performance.

# The Age of Free Performance is Over

Moore's law is not giving free performance any more

We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."
— Don Knuth

Performance is the currency of computing. You can often "buy" needed properties with performance.



Performance Hints: https://abseil.io/fast/hints.html

# The Age of Free Performance is Over

Two ways to get better performance:

1. Remove software abstractions costs

2. Build domain-specific hardware

Both requires specialization. A compiler is

a generator of specialized code.

# Inefficient abstractions mechanisms in software and inefficient use of hardware

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction,  except the problems of too many levels of abstraction.”*
- David Wheeler

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*"Abstraction is selective ignorance"*
- Andrew Koenig

*"All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction."*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |

Figure from "There's plenty of room at the Top: What will drive computer performance after Moore's law?" Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*"Abstraction is selective ignorance"*
- Andrew Koenig

*"All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction."*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |

Figure from "There's plenty of room at the Top: What will drive computer performance after Moore's law?" Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*"Abstraction is selective ignorance"*
- Andrew Koenig

*"All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction."*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |

Figure from "There's plenty of room at the Top: What will drive computer performance after Moore's law?" Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |
| 7 | plus AVX intrinsics | 0.41 | 337.812 | 62,806 | 2.7 | 40.45 |

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*"Abstraction is selective ignorance"*
- Andrew Koenig

*"All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction."*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |
| 7 | plus AVX intrinsics | 0.41 | 337.812 | 62,806 | 2.7 | 40.45 |

17h vs 1s!

Figure from "There's plenty of room at the Top: What will drive computer performance after Moore's law?" Leiserson et al., Science 368

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |
| 7 | plus AVX intrinsics | 0.41 | 337.812 | 62,806 | 2.7 | 40.45 |

17h vs 1s!

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

Parallelism

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |
| 7 | plus AVX intrinsics | 0.41 | 337.812 | 62,806 | 2.7 | 40.45 |

17h vs 1s!

Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

Parallelism

Locality

# Inefficient abstractions mechanisms in software and inefficient use of hardware

*“Abstraction is selective ignorance”*
- Andrew Koenig

*“All problems in computer science can be solved by another level of abstraction, except the problems of too many levels of abstraction.”*
- David Wheeler

**Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices.** Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

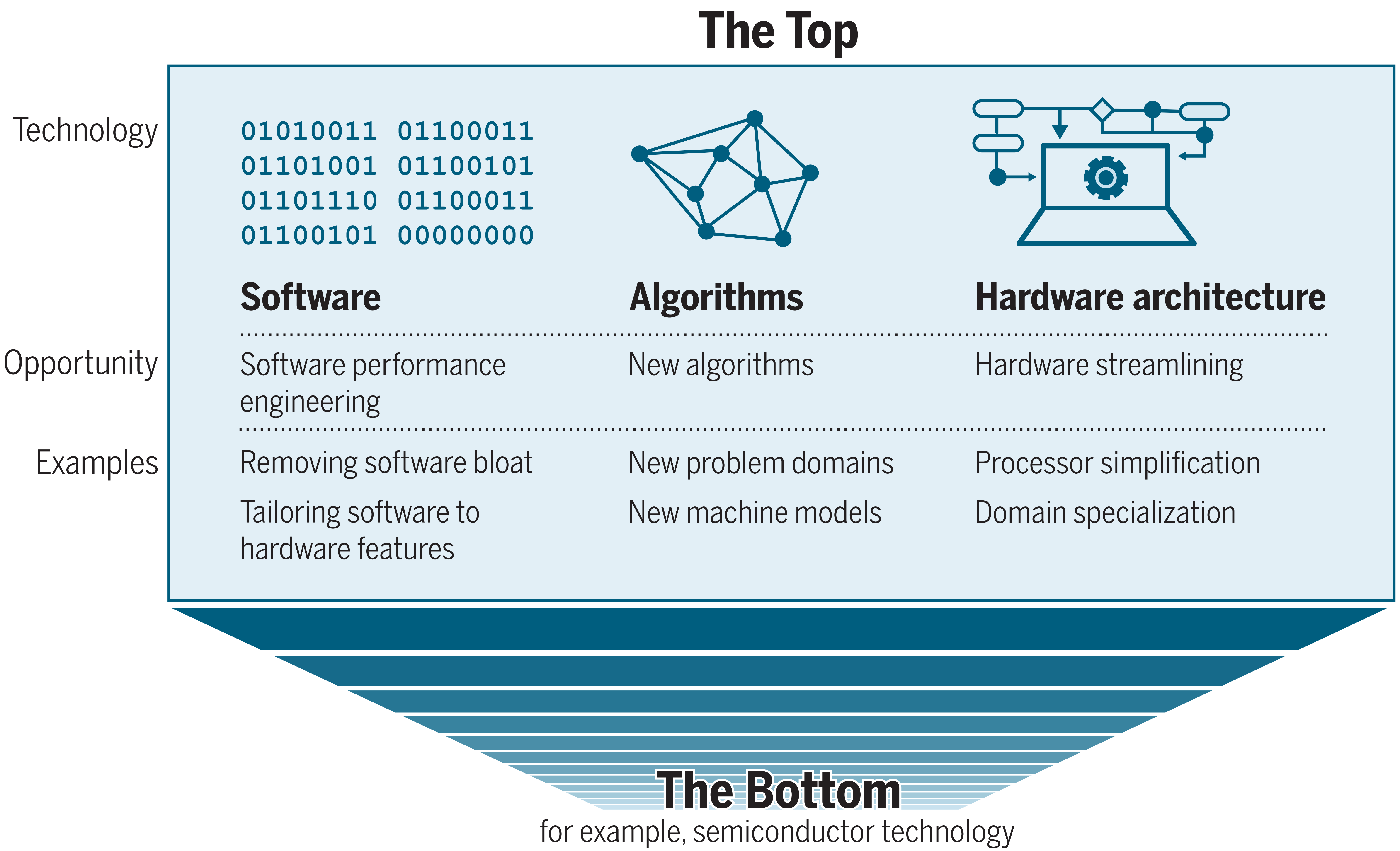
| Version | Implementation | Running time (s) | GFLOPS | Absolute speedup | Relative speedup | Fraction of peak (%) |
|---|---|---|---|---|---|---|
| 1 | Python | 25,552.48 | 0.005 | 1 | — | 0.00 |
| 2 | Java | 2,372.68 | 0.058 | 11 | 10.8 | 0.01 |
| 3 | C | 542.67 | 0.253 | 47 | 4.4 | 0.03 |
| 4 | Parallel loops | 69.80 | 1.969 | 366 | 7.8 | 0.24 |
| 5 | Parallel divide and conquer | 3.80 | 36.180 | 6,727 | 18.4 | 4.33 |
| 6 | plus vectorization | 1.10 | 124.914 | 23,224 | 3.5 | 14.96 |
| 7 | plus AVX intrinsics | 0.41 | 337.812 | 62,806 | 2.7 | 40.45 |

17h vs 1s!

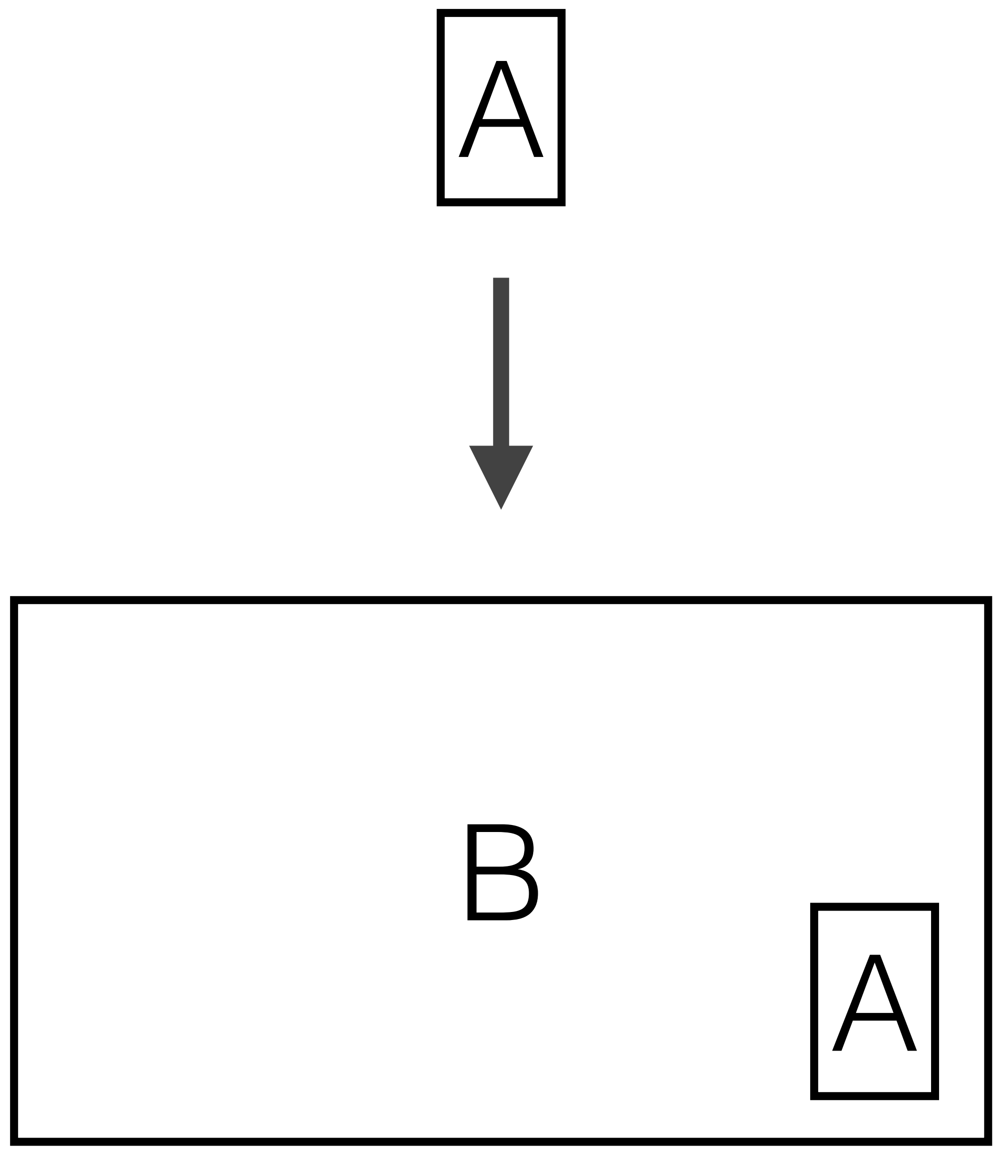
Figure from “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” Leiserson et al., Science 368

Parallelism Locality Specialization

# There's plenty of room at the top




## Problem reduction



# Abstraction with friction from traditional library composition

$$A = B \odot (CD)$$

Traditional Library Composition

```
T = matmul(C, D);
A =  elmul(B, T);
```

Three pitfalls:

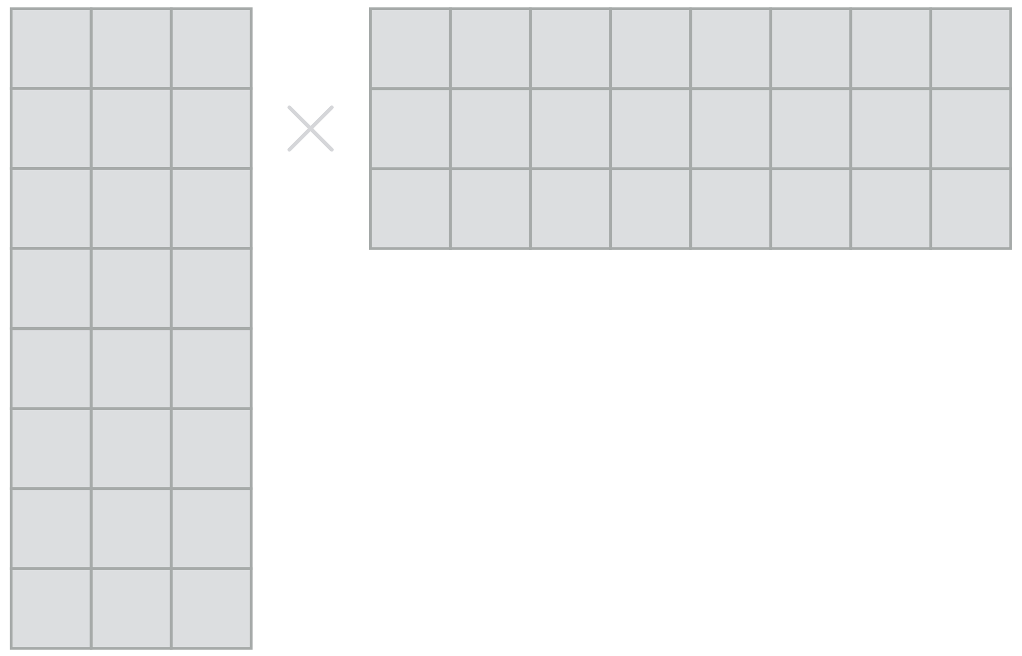
1. Loose temporal locality
2. Data structures must match what functions expect
3. May cause asymptotic slow-down

# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

$$A = B \odot (CD)$$

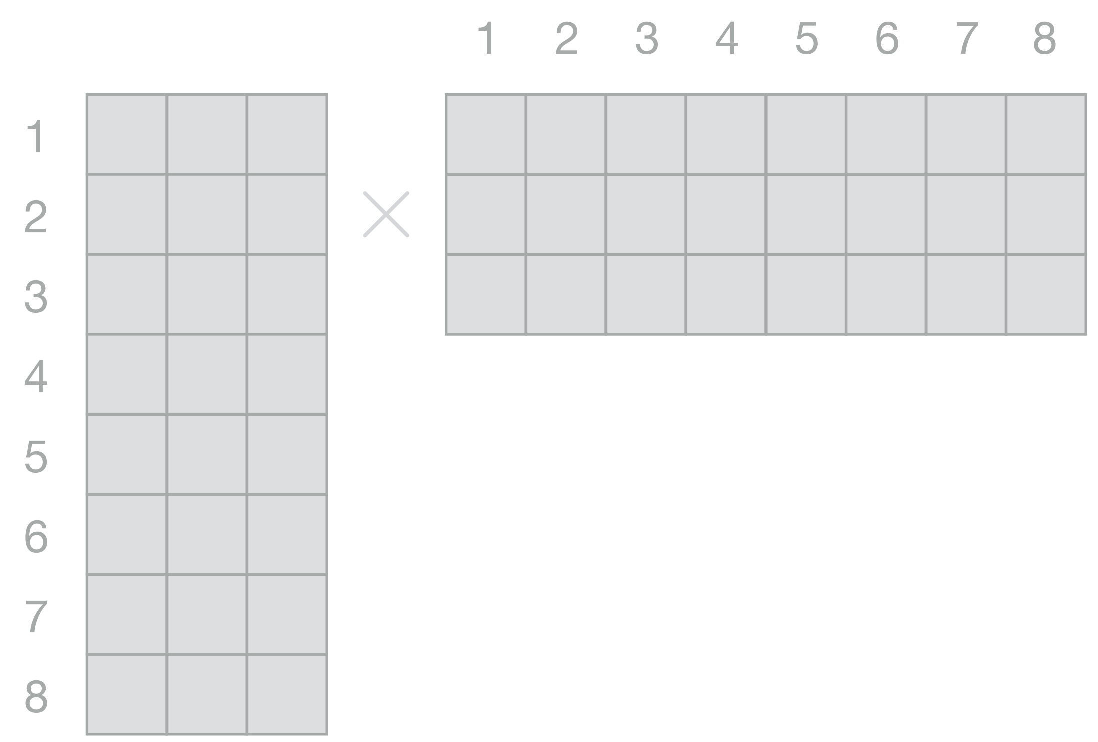
# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

$$A = B \odot (CD)$$



# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

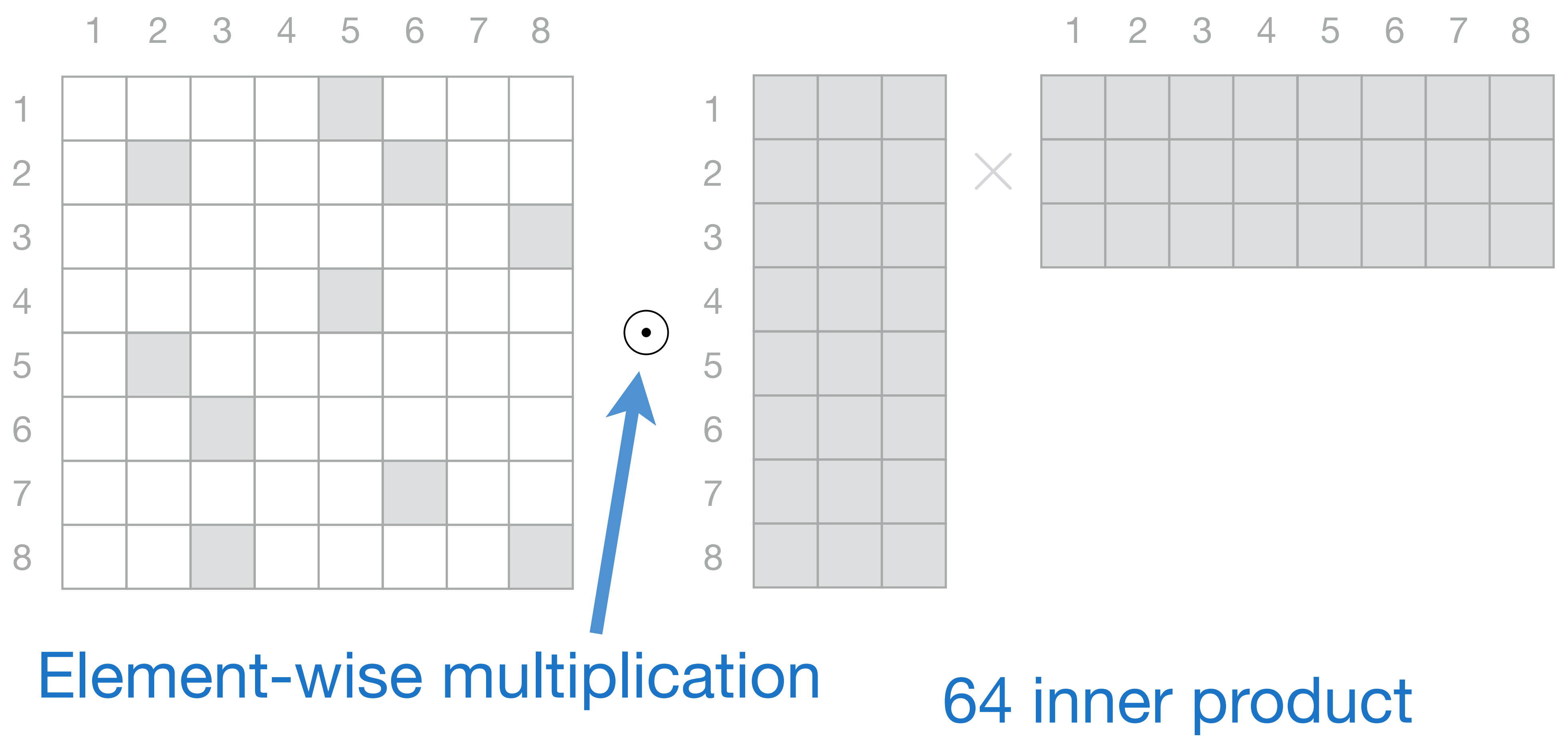
$$A = B \odot (CD)$$




64 inner product

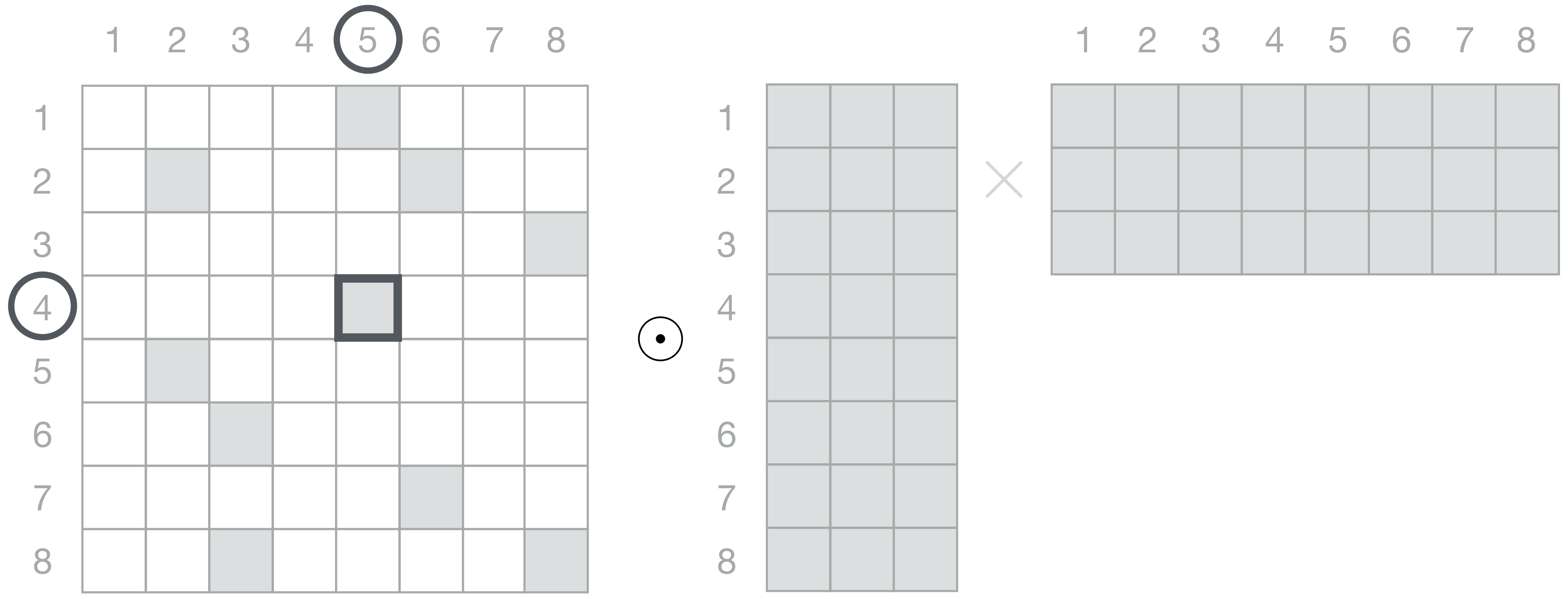
# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

$$A = B \odot (CD)$$



# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

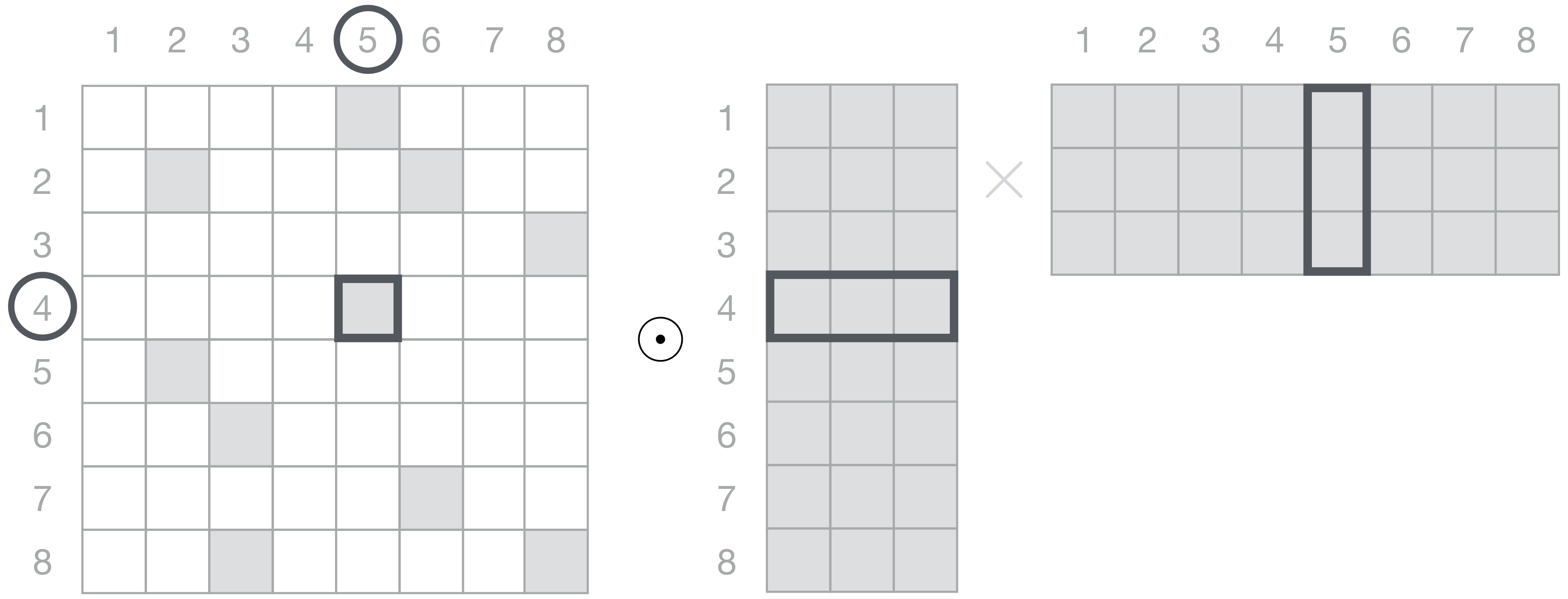
$$A = B \odot (CD)$$




64 inner product

# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

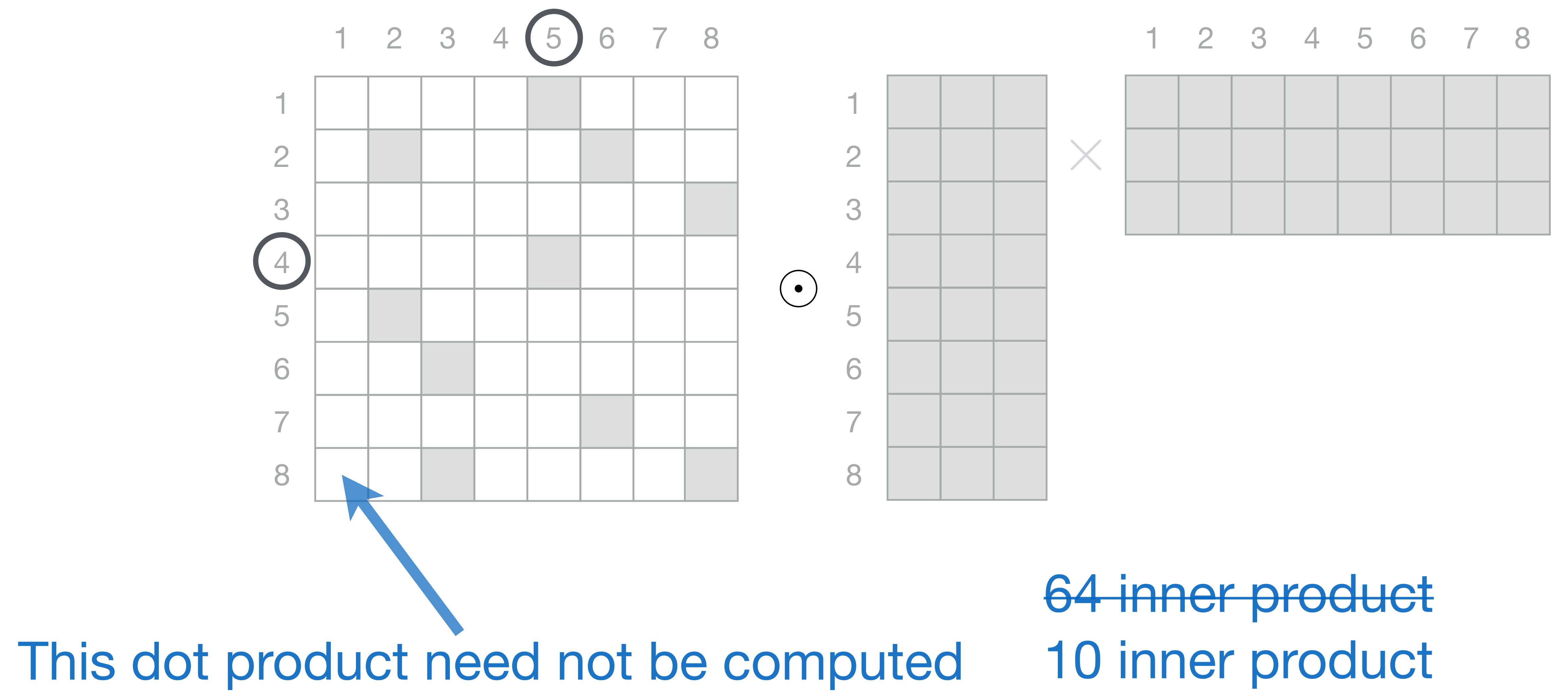
$$A = B \odot (CD)$$




64 inner product

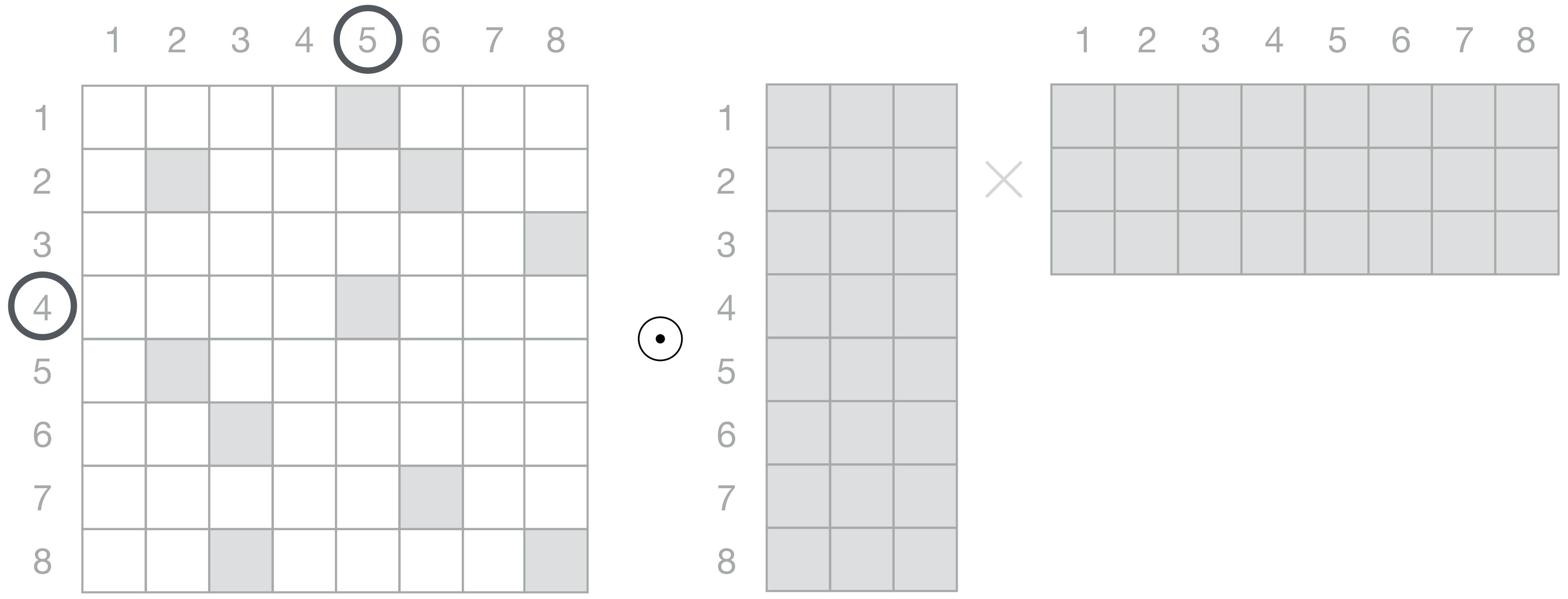
# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

$$A = B \odot (CD)$$



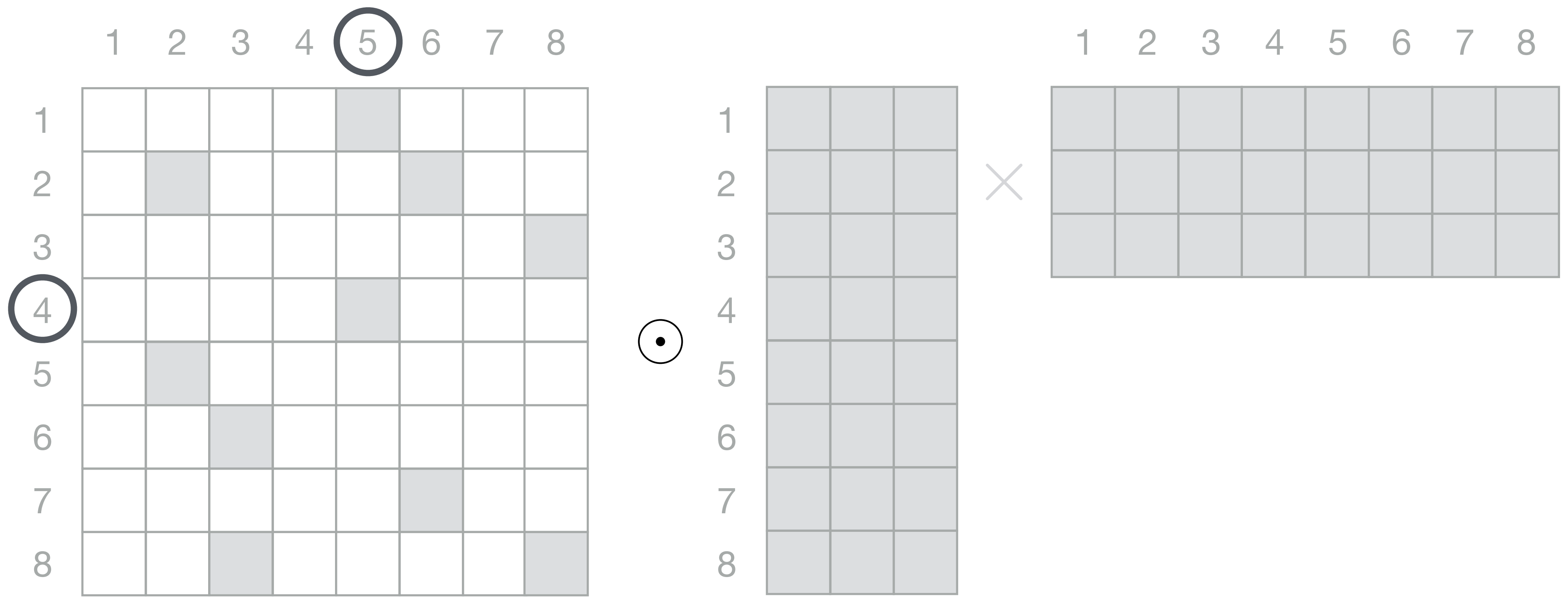
# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

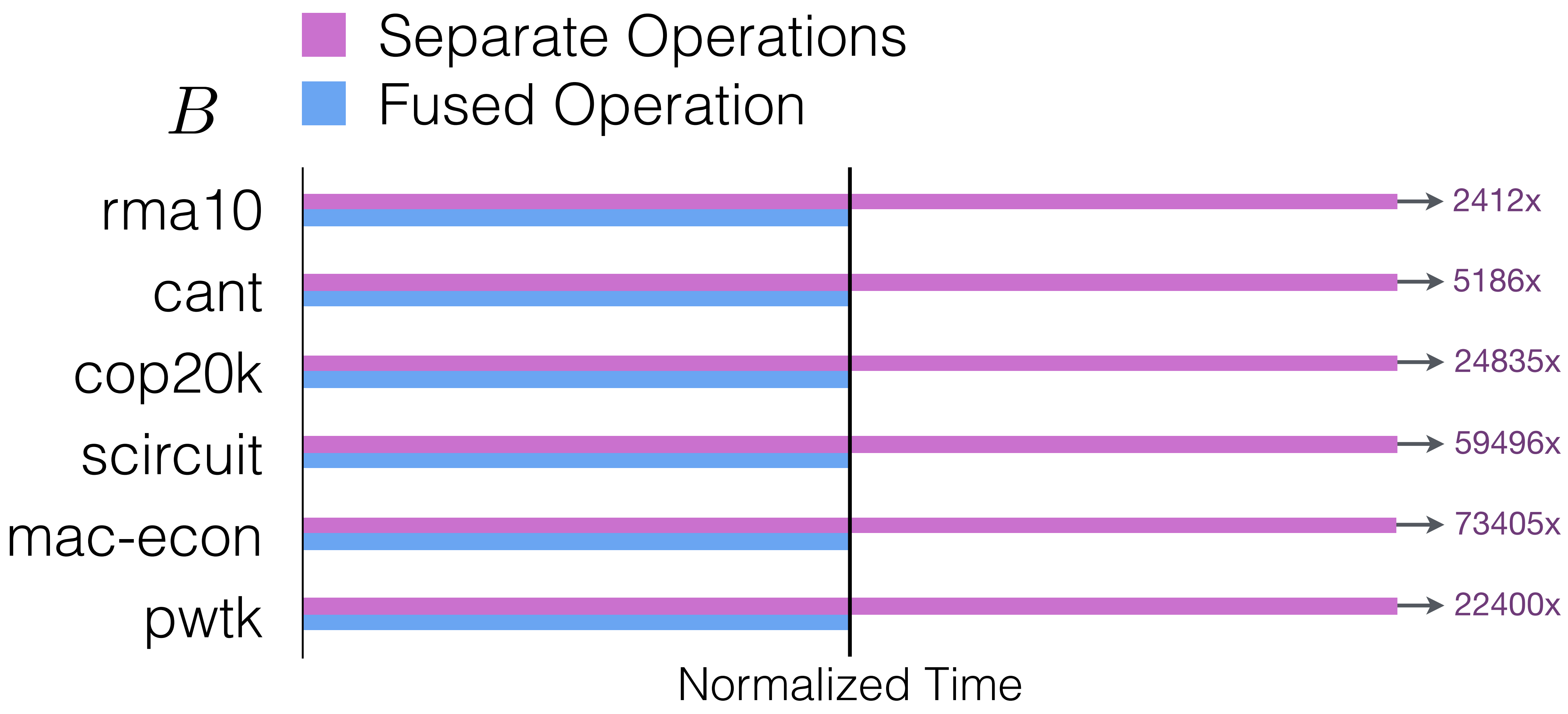
$$A = B \odot (CD)$$



# Example 1: Sampled Dense-Dense Matrix Multiplication with Linear Algebra

$$A = B \odot (CD)$$





# Example 2: Triangle Query with Relational Algebra

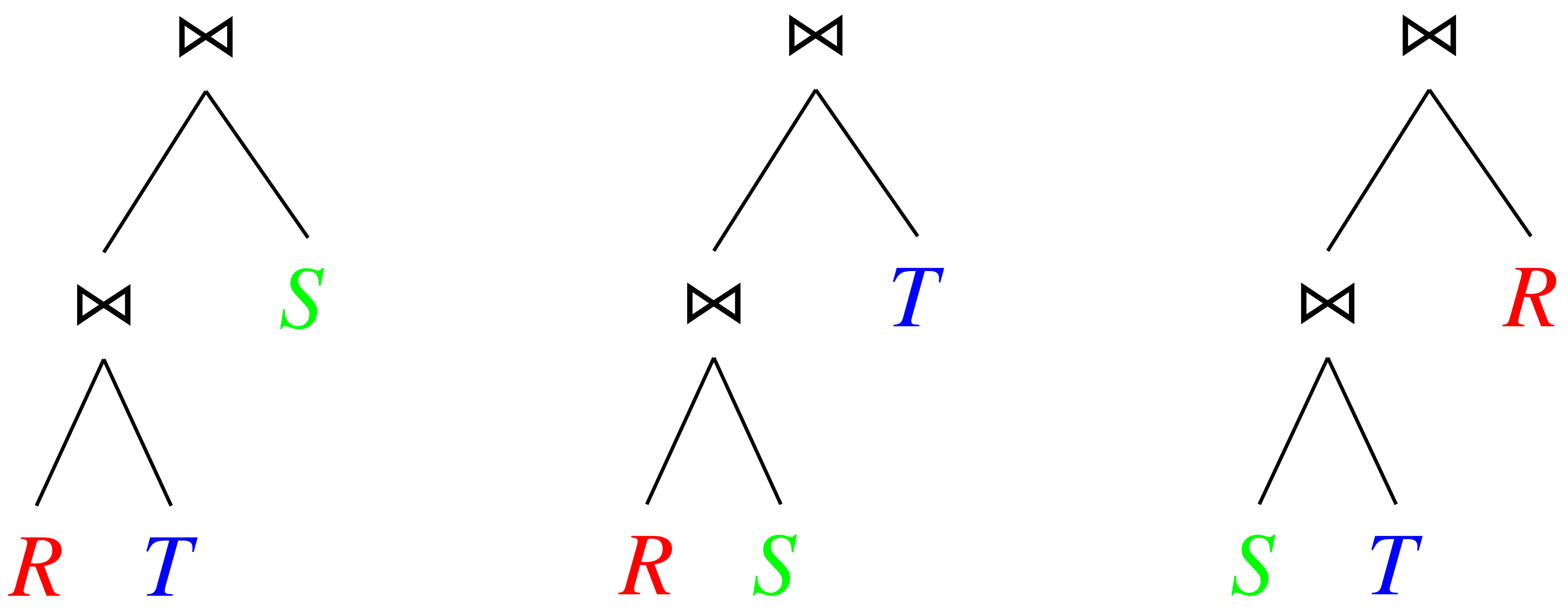
$$Q_{\triangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C)$$

# Example 2: Triangle Query with Relational Algebra

$$Q_\triangle = R(A,B) \bowtie S(B,C) \bowtie T(A,C) \qquad O(N^{3/2})$$

# Example 2: Triangle Query with Relational Algebra

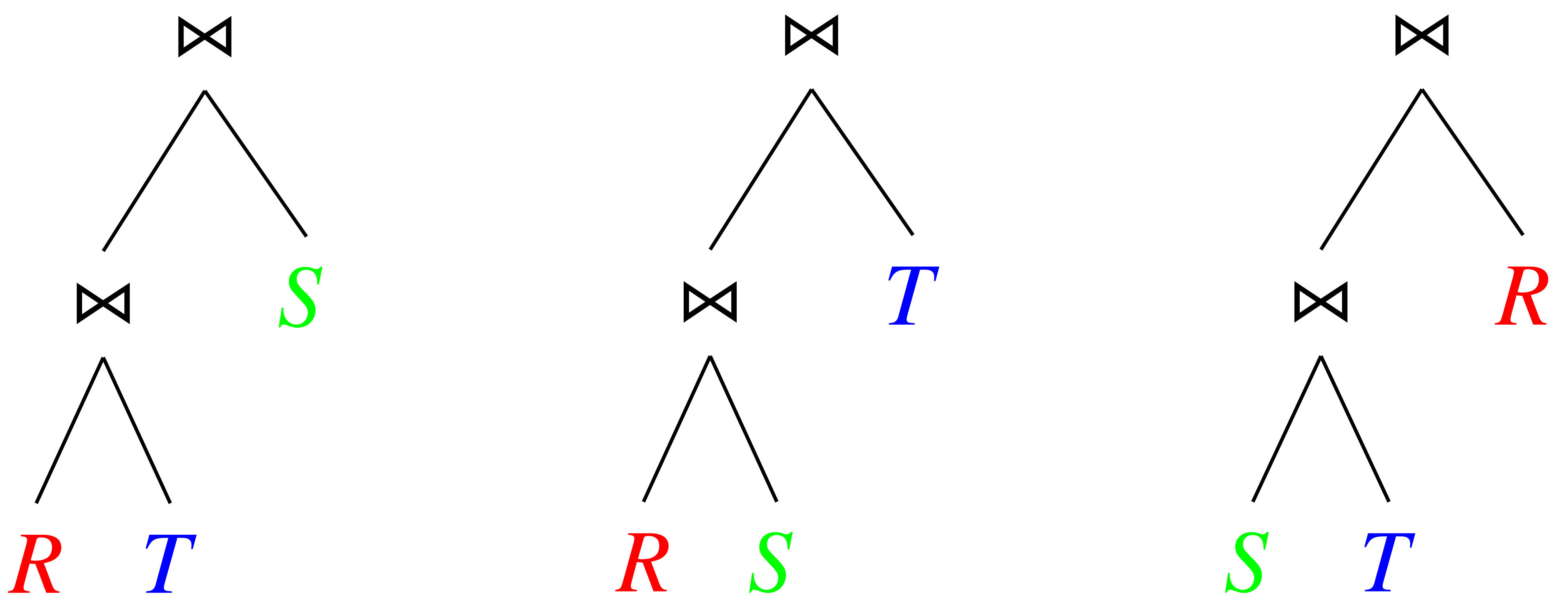
$$Q_{\triangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C) \qquad O(N^{3/2})$$




$O(N^2)$

# Example 2: Triangle Query with Relational Algebra

$$Q_{\triangle} = R(A,B) \bowtie S(B,C) \bowtie T(A,C) \qquad O(N^{3/2})$$




$O(N^2)$

**Algorithm 2** Computing $Q_{\triangle}$ by delaying computation.

**Input:** $R(A,B), S(B,C), T(A,C)$ in sorted order

$Q \leftarrow \emptyset$
$L_A \leftarrow \pi_A(R) \cap \pi_A(T)$
**For each** $a \in L_A$ **do**
  $L_B^a \leftarrow \pi_B(\sigma_{A=a}(R)) \cap \pi_B(S)$
  **For each** $b \in L_B^a$ **do**
    $L_C^{a,b} \leftarrow \pi_C(\sigma_{B=b}(S)) \cap \pi_C(\sigma_{A=a}(T))$
    **For each** $c \in L_C^{a,b}$ **do**
      Add $(a,b,c)$ to $Q$
**Return** $Q$

Figures from Ngo, Ré and Rudra (2013), with algorithm from Veldhuizen (2014)

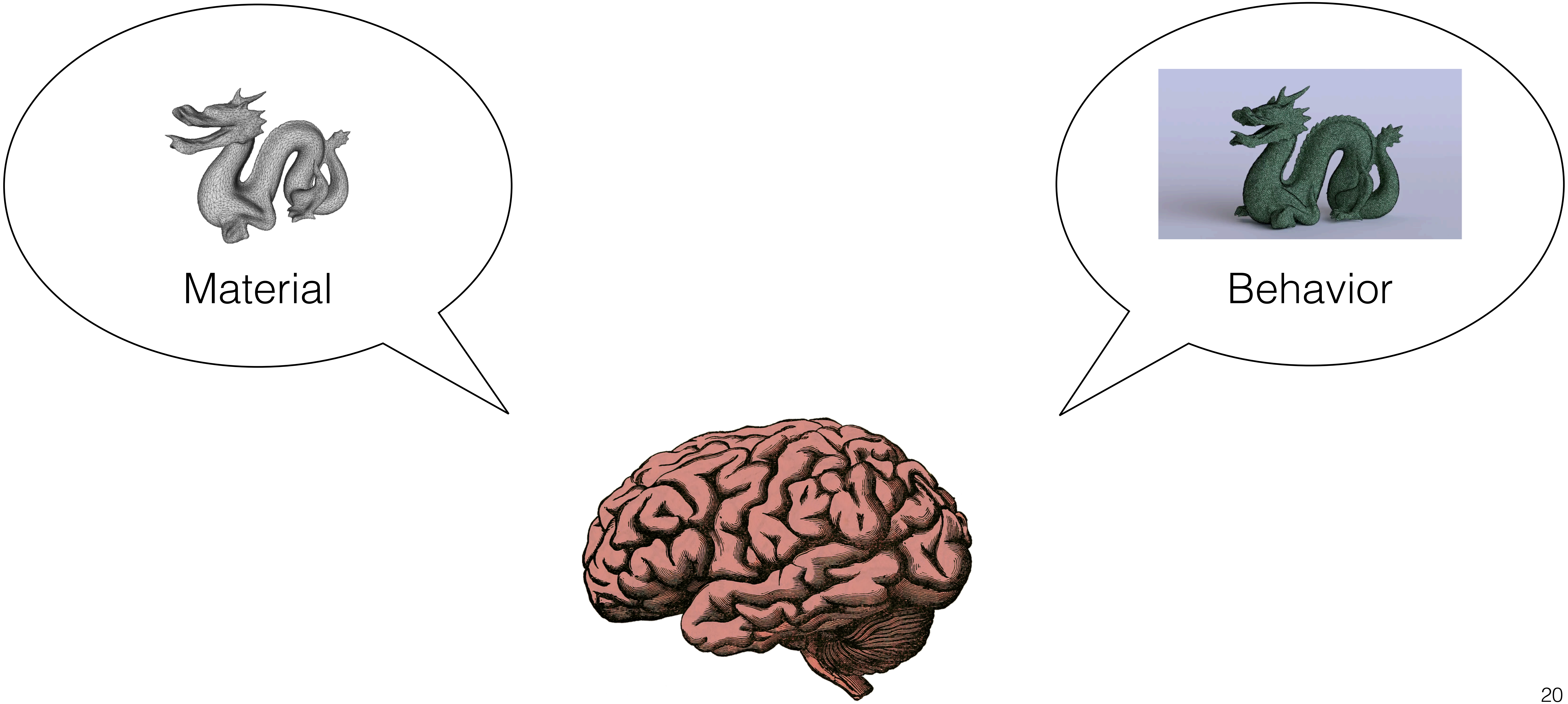
$O(N^{3/2})$

# Example 3: Simulation with Meshes and Linear Algebra

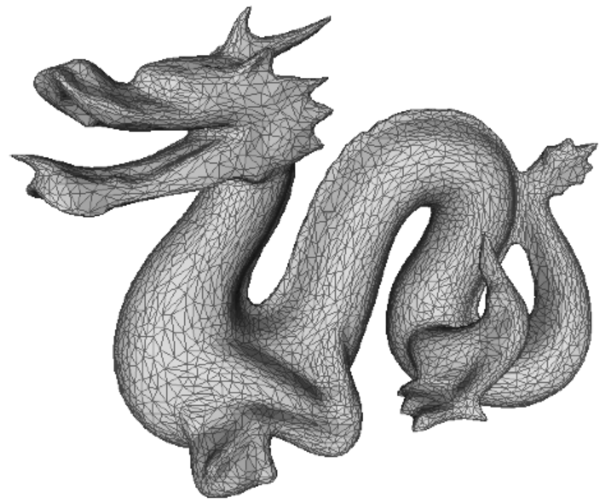
# Example 3: Simulation with Meshes and Linear Algebra

Behavior

# Example 3: Simulation with Meshes and Linear Algebra



# Example 3: Simulation with Meshes and Linear Algebra

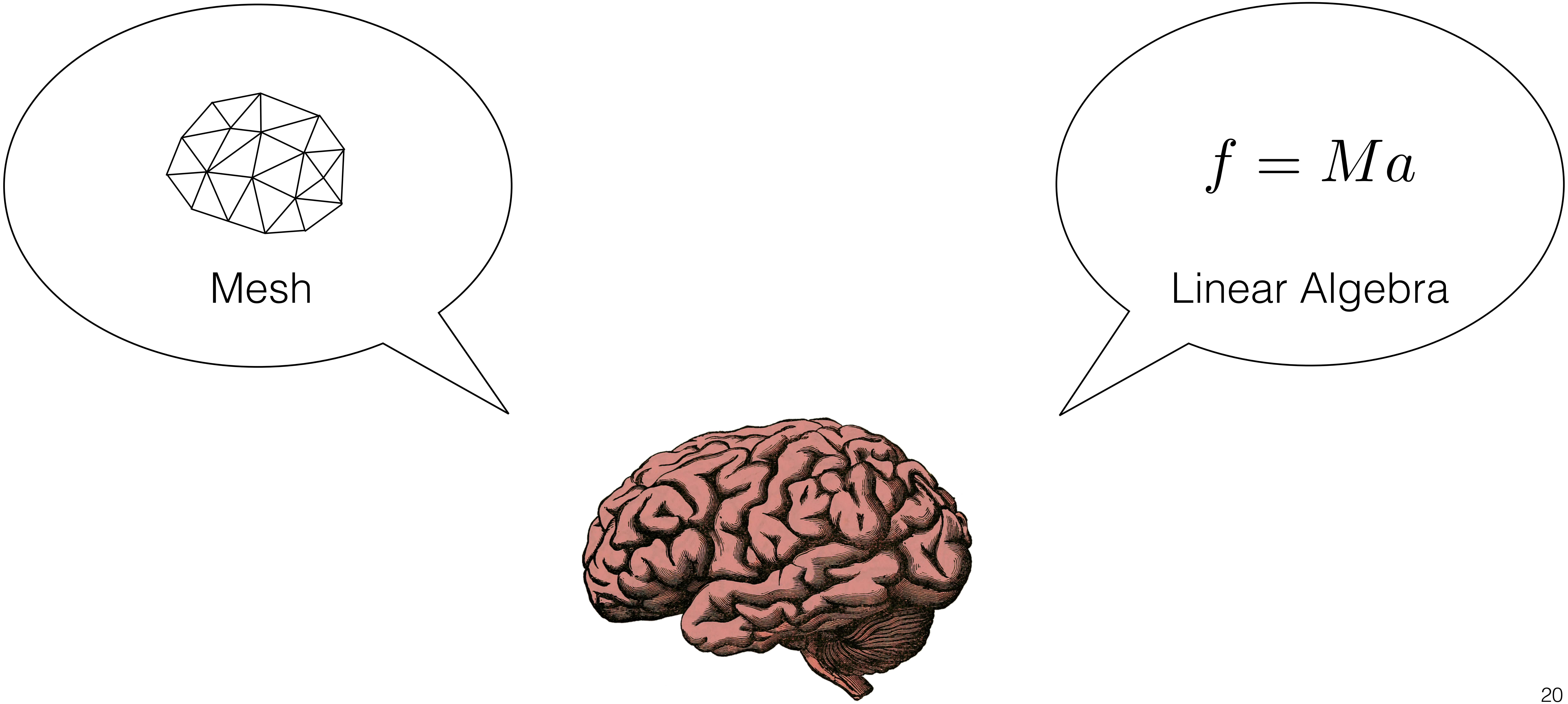


Material

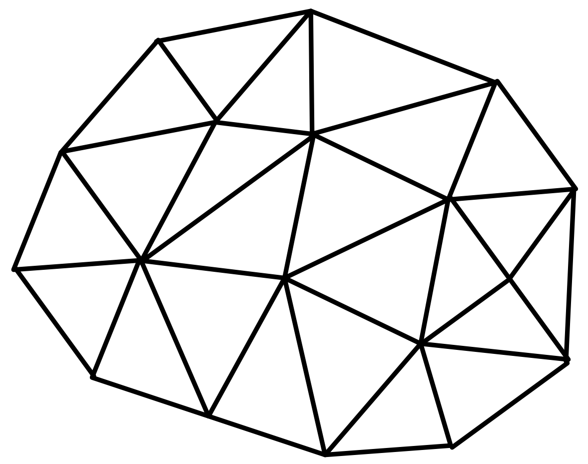
$$f = Ma$$

Linear Algebra

# Example 3: Simulation with Meshes and Linear Algebra



# Example 3: Simulation with Meshes and Linear Algebra



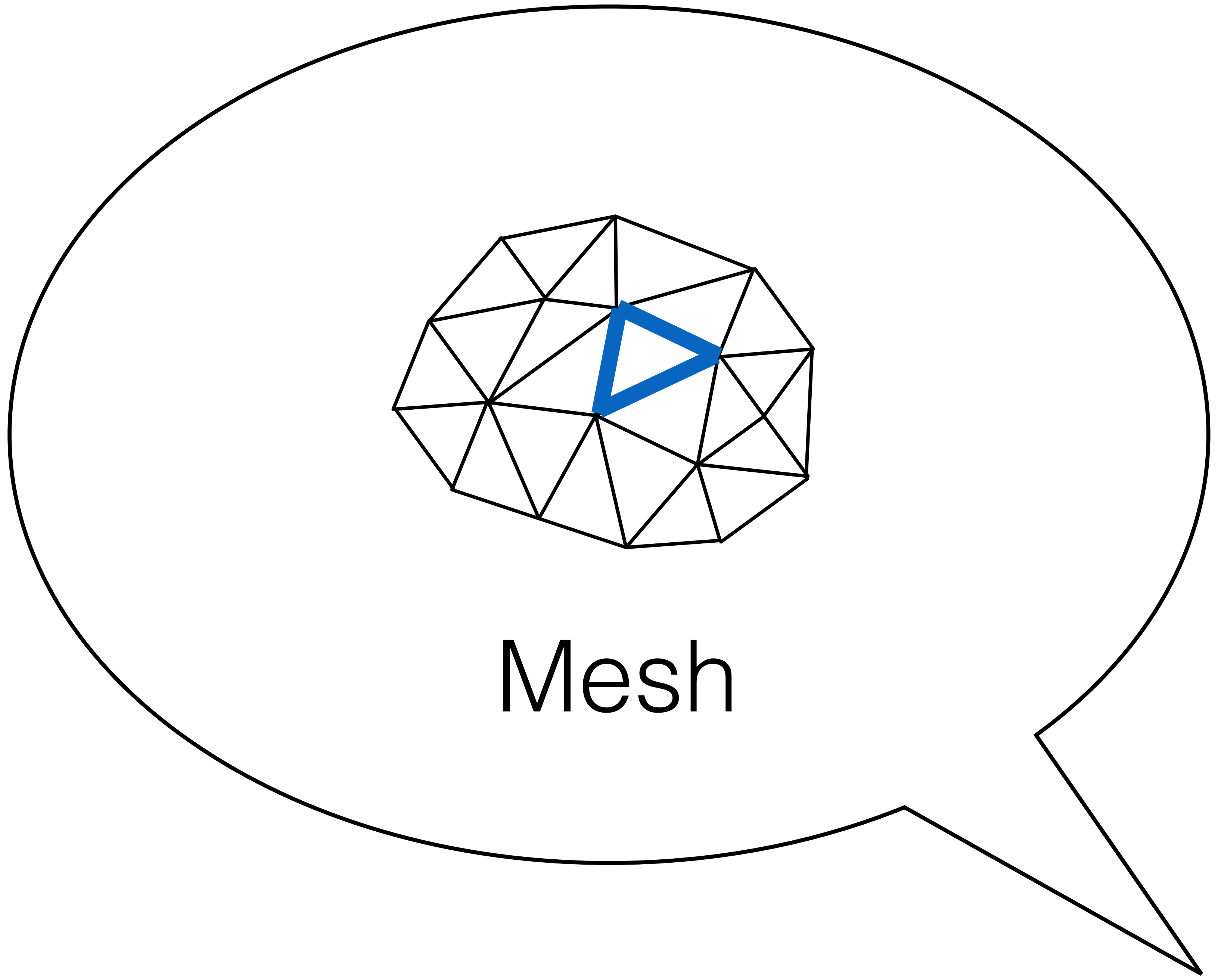
Mesh

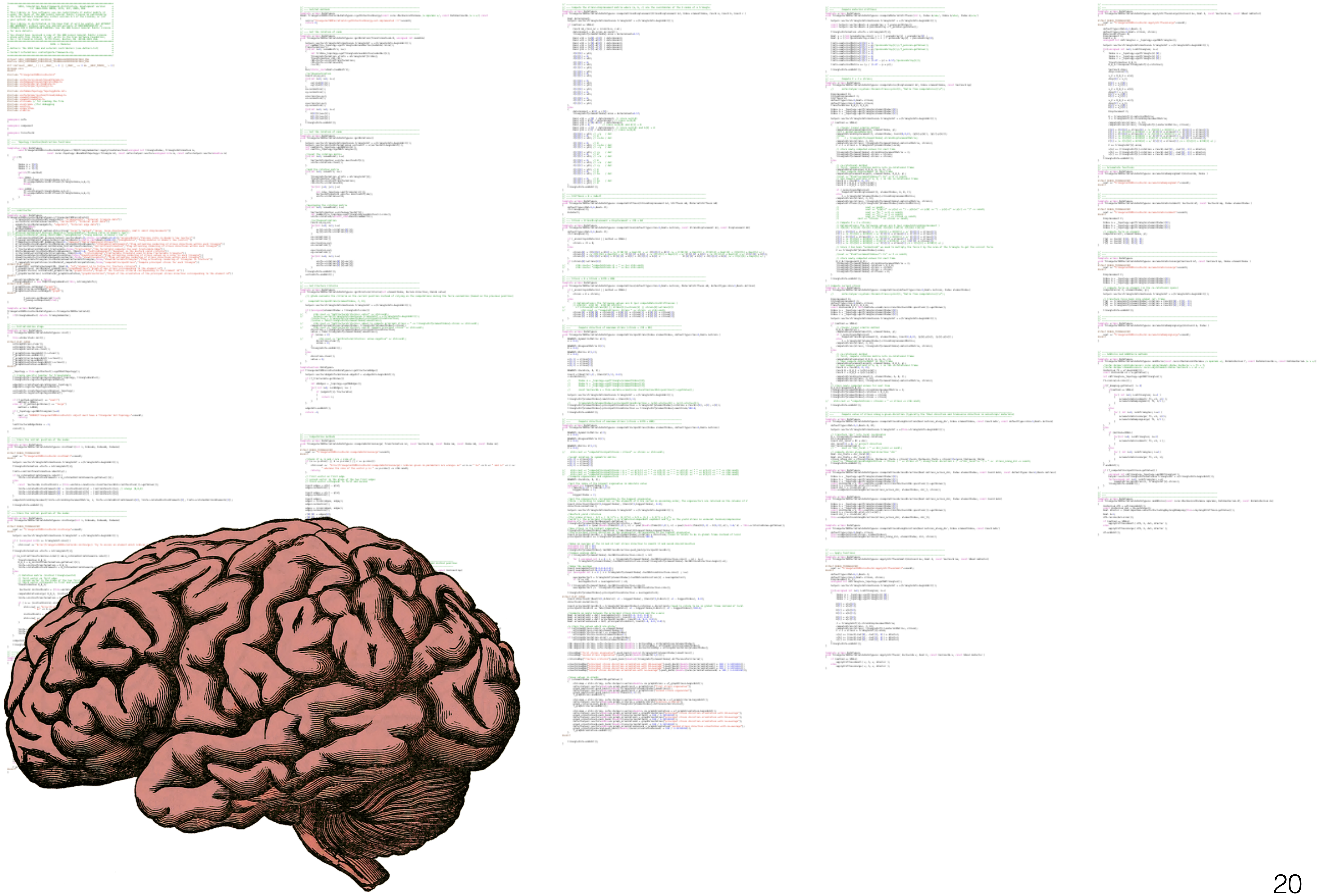

$$f = Ma$$

Linear Algebra

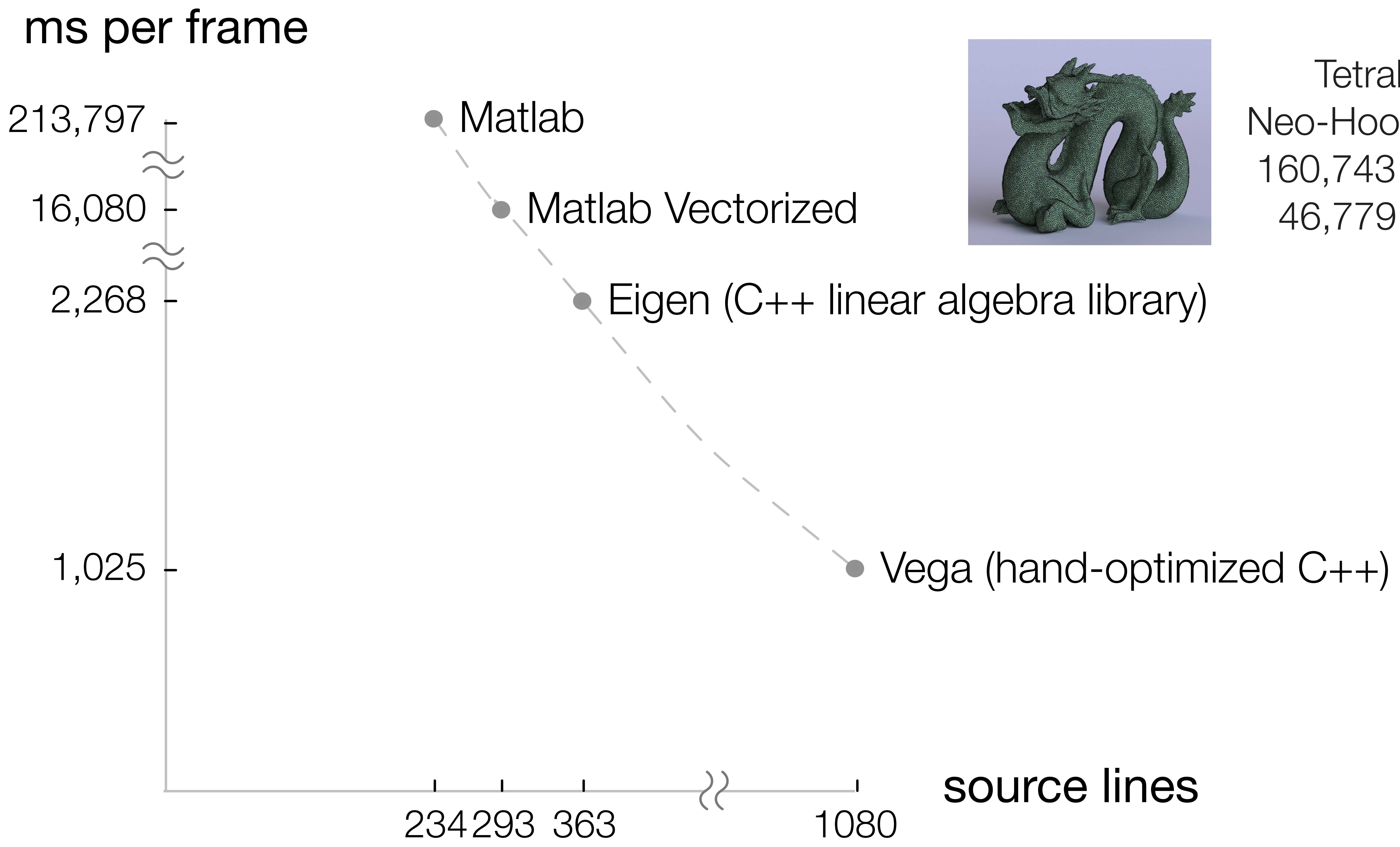
# Example 3: Simulation with Meshes and Linear Algebra

## Matrix-free in-place stencil computation



# Example 3: Simulation with Graphs and Linear Algebra




Tetrahedral
Neo-Hookean FEM
160,743 elements
46,779 vertices

Figure from Kjolstad et al (2016)

# Example 3: Simulation with Graphs and Linear Algebra

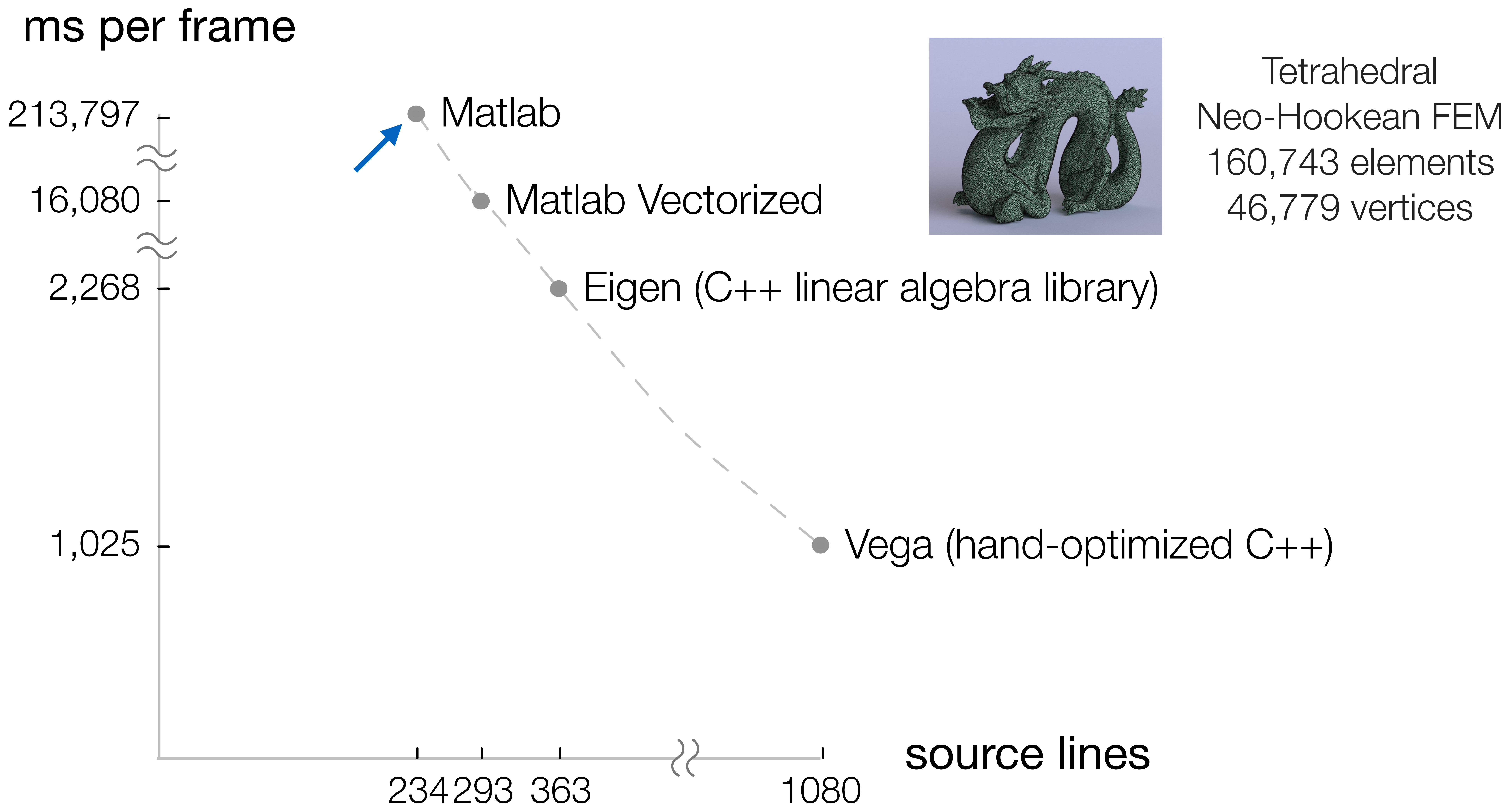



Figure from Kjolstad et al (2016)

# Example 3: Simulation with Graphs and Linear Algebra

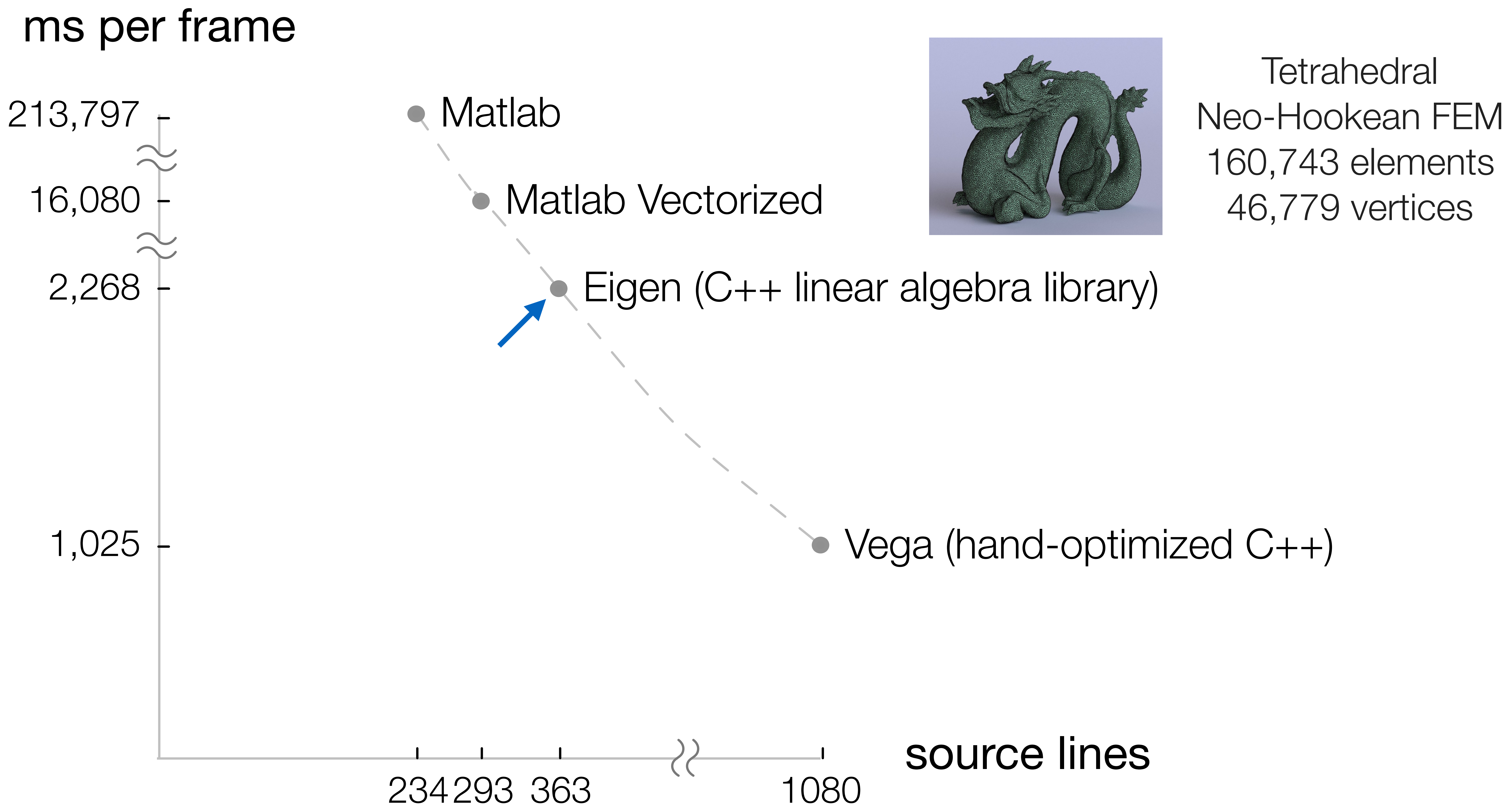
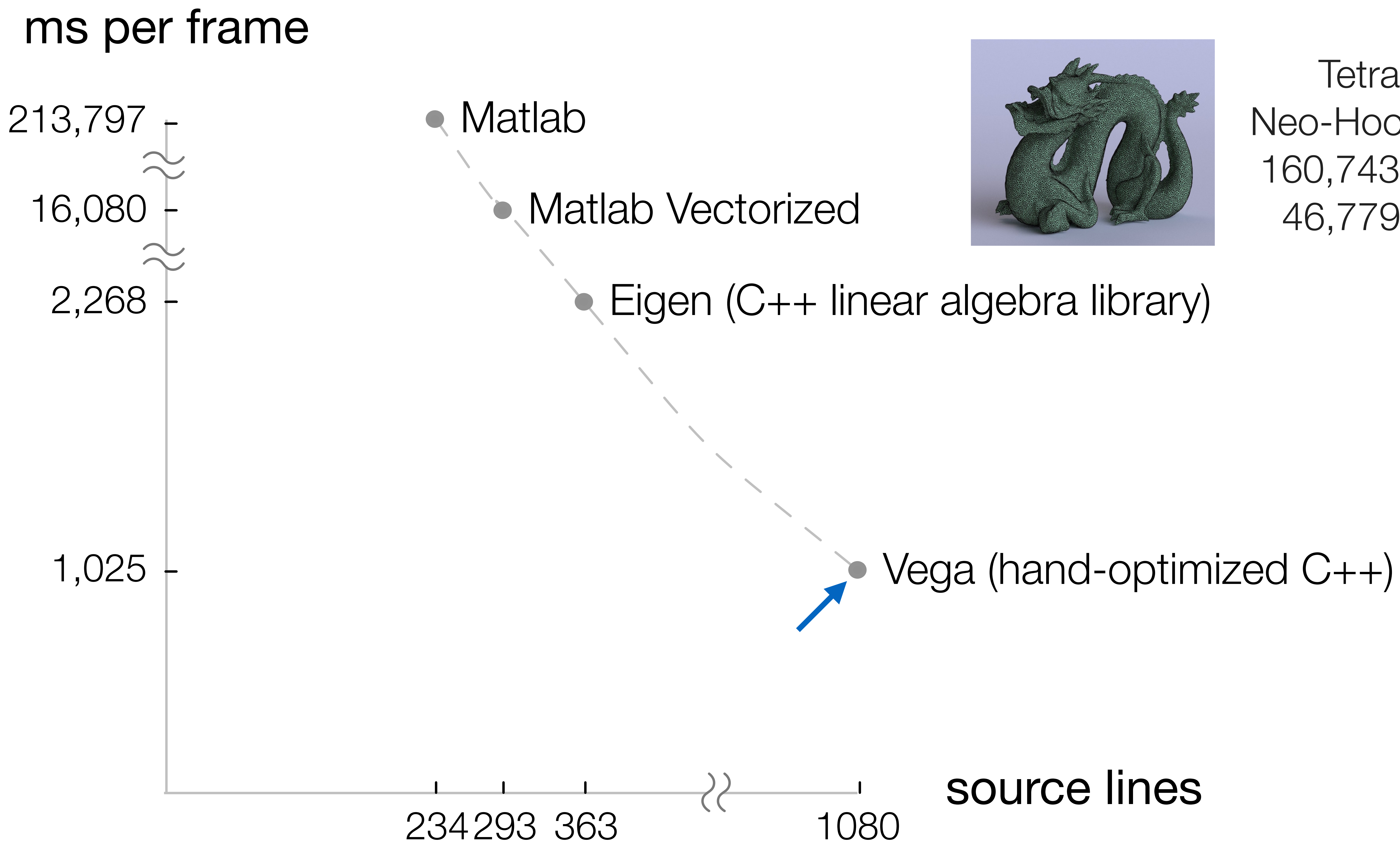



Figure from Kjolstad et al (2016)

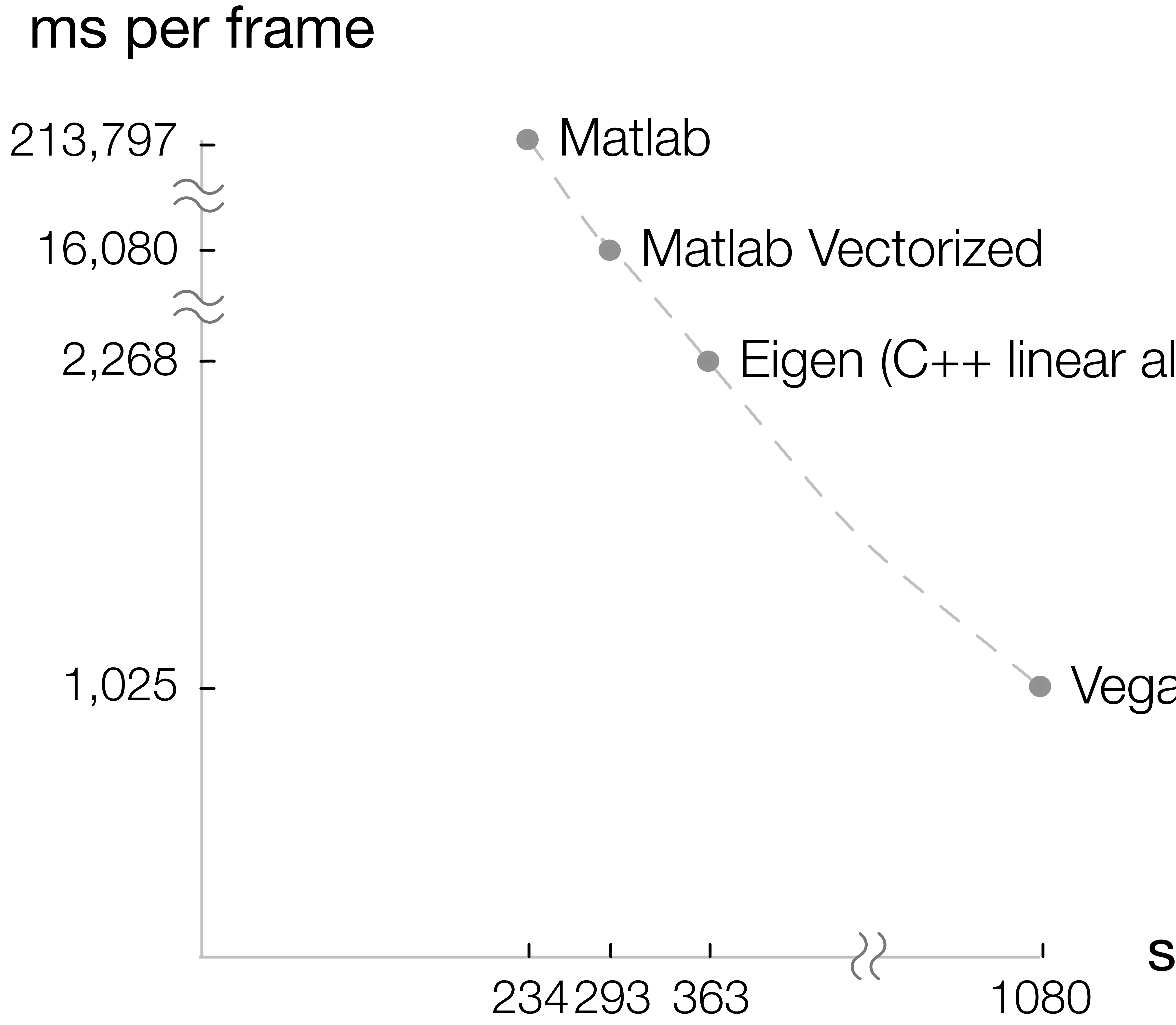
# Example 3: Simulation with Graphs and Linear Algebra

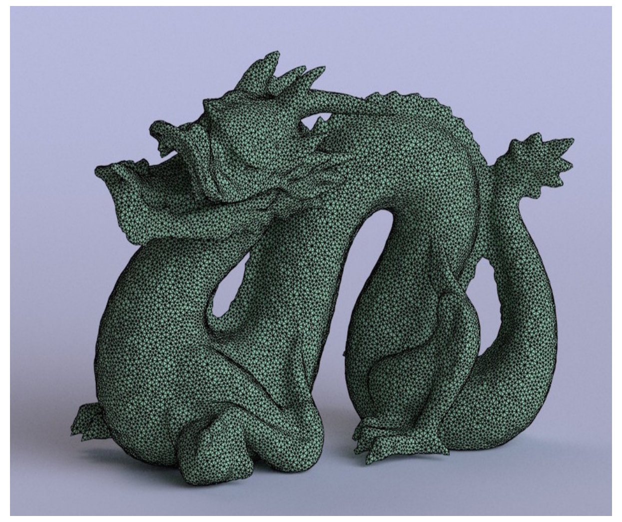



Tetrahedral
Neo-Hookean FEM
160,743 elements
46,779 vertices

Figure from Kjolstad et al (2016)

# Example 3: Simulation with Graphs and Linear Algebra



Tetrahedral
Neo-Hookean FEM
160,743 elements
46,779 vertices

Figure from Kjolstad et al (2016)

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $\quad A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $\quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

Linear Algebra

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

# Too many combinations for a fixed-function library

$$a = Bc$$

$$a = Bc + a$$

$$a = Bc + b \qquad A = B + C \qquad a = \alpha Bc + \beta a$$

$$a = B^T c \qquad A = \alpha B \qquad a = B(c + d)$$

$$a = B^T c + d \qquad A = B + C + D \qquad A = BC$$

$$A = B \odot C \qquad a = b \odot c \qquad A = 0 \qquad A = B \odot (CD)$$

$$A = BCd \qquad A = B^T \qquad a = B^T Bc$$

$$a = b + c \qquad A = B \qquad K = A^T CA$$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

Data analytics
(tensor factorization)

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

# Too many combinations for a fixed-function library

$$a = Bc$$

$$a = Bc + a$$

$$a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a$$

$$a = B^T c \quad A = \alpha B \quad a = B(c + d)$$

$$a = B^T c + d \quad A = B + C + D \quad A = BC$$

$$A = B \odot C \quad a = b \odot c \quad A = 0 \quad A = B \odot (CD)$$

$$A = BCd \quad A = B^T \quad a = B^T Bc$$

$$a = b + c \quad A = B \quad K = A^T CA$$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \quad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \quad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

Quantum Chromodynamics

# Too many combinations for a fixed-function library

CSparse

$a = Bc + a$

Eigen (SpMV)

$a = Bc$

OSKI

$a = \alpha Bc + \beta a$

$a = Bc + b$

$A = B + C$

PETSc

$a = B^T c$

$A = \alpha B$

$a = B(c + d)$

$a = B^T c + d$

$A = B + C + D$

$A = BC$

$A = B \odot C$

$a = b \odot c$

$A = 0$

$A = B \odot (CD)$

$A = BCd$

$A = B^T$

$a = B^T Bc$

$a = b + c$

$A = B$

$K = A^T CA$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

# Too many combinations for a fixed-function library

CSparse

Eigen (SpMV)

$a = Bc$

$a = Bc + a$

OSKI

OSKI has 282 specialized variants of this expression

$a = Bc + b$ $\quad A = B + C \quad a = \alpha Bc + \beta a$

PETSc $\quad a = B^T c \quad A = \alpha B \quad a = B(c+d)$

$a = B^T c + d \quad A = B + C + D \quad A = BC$

$A = B \odot C \quad a = b \odot c \quad A = 0 \quad A = B \odot (CD)$

$A = BCd \quad A = B^T \quad a = B^T Bc$

$a = b + c \quad A = B \quad K = A^T CA$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $A = B + C$ $a = \alpha Bc + \beta a$

$a = B^T c$ $A = \alpha B$ $a = B(c + d)$

$a = B^T c + d$ $A = B + C + D$ $A = BC$

$A = B \odot C$ $a = b \odot c$ $A = 0$ $A = B \odot (CD)$

$A = BCd$ $A = B^T$ $a = B^T Bc$

$a = b + c$ $A = B$ $K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\,\overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\,\overline{P_{ip}}$

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $A = B + C$ $a = \alpha Bc + \beta a$

$a = B^T c$ $A = \alpha B$ $a = B(c + d)$

$a = B^T c + d$ $A = B + C + D$ $A = BC$

$A = B \odot C$ $a = b \odot c$ $A = 0$ $A = B \odot (CD)$

$A = BCd$ $A = B^T$ $a = B^T Bc$

$a = b + c$ $A = B$ $K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

Dense Matrix

Thermal Simulation CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$$

$$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \qquad A_{ij} = \sum_{k} B_{ijk} c_k$$

$$A_{ijk} = \sum_{l} B_{ikl} C_{lj} \qquad A_{ik} = \sum_{j} B_{ijk} c_j$$

$$A_{jk} = \sum_{i} B_{ijk} c_i \qquad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$$

$$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$$

$$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$$

$$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$$

$\times$

Web matrix [BG 2008]

Dense Matrix

Thermal Simulation — CSR

DCSR

BCSR — Finite Elements Method, Block-Sparse NN Weights [GRK 2017]

Data Analytics — COO

ELLPACK

CSB

Blocked COO

CSC — Mesh Simulations on GPUs [BG 2009]

Convolutions — DIA

Blocked DIA

DCSC

Eulerian Simulations

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $\quad A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $\quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_i z_i (\sum_j z_j \theta_{ij})(\sum_k z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $\quad A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $\quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

Coordinates

Dense Tensors

CSF

Blocked Tensors

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $\quad A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $\quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

Coordinates

Dense Tensors

CSF

Blocked Tensors

Linked Lists Database

Compression Schemes

Cloud Storage

# Too many combinations for a fixed-function library

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $\quad A = B + C$ $\quad a = \alpha Bc + \beta a$

$a = B^T c$ $\quad A = \alpha B$ $\quad a = B(c + d)$

$a = B^T c + d$ $\quad A = B + C + D$ $\quad A = BC$

$A = B \odot C$ $\quad a = b \odot c$ $\quad A = 0$ $\quad A = B \odot (CD)$

$A = BCd$ $\quad A = B^T$ $\quad a = B^T Bc$

$a = b + c$ $\quad A = B$ $\quad K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $\quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $\quad A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $\quad A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $\quad A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

Dense Matrix

CSR DCSR BCSR

COO ELLPACK CSB

Blocked COO CSC

DIA Blocked DIA DCSC

Sparse vector Hash Maps

Coordinates

Dense Tensors

CSF

Blocked Tensors

Linked Lists Database

Compression Schemes

Cloud Storage

$\times$

CPU

GPUs TPUs

FPGA

Sparse Tensor Hardware

Cloud Computers

Supercomputers

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k B_{ijk} c_k$$

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k B_{ijk} c_k$$

dense dense

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k \underbrace{B_{ijk}}_{\text{dense}} \underbrace{c_k}_{\text{dense}}$$

```
for (int i = 0; i < m; i++) {

  for (int j = 0; j < n; j++) {
    int pB2 = i*n + j;
    int pA2 = i*n + j;
    double t = 0.0;
    for (int k = 0; k < o; k++) {
      int pB3 = pB2*o + k;
      t += B[pB3] * c[k];
    }
    A[pA2] = t;
  }
}
```

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k B_{ijk} c_k$$

CSF dense

```
for (int pA = 0; pA < m*n; pA++) {
  A[pA] = 0.0;
}
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_crd[pB2];
    int pA2 = i*n + j;
    double t = 0.0;
    for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
      int k = B3_crd[pB3];
      t += B[pB3] * c[k];
    }
    A[pA2] = t;
  }
}
```

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k B_{ijk} c_k$$

CSF compressed

```
for (int pA = 0; pA < m*n; pA++) {
  A[pA] = 0.0;
}
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_crd[pB2];
    int pA2 = i*n + j;
    double t = 0.0;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
      int kB = B3_crd[pB3];
      int kc = c1_crd[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        t += B[pB3] * c[pc1];
      }
      pB3 += (int)(kB == k);
      pc1 += (int)(kc == k);
    }
    A[pA2] = t;
  }
}
```

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ij} = \sum_k \underset{\uparrow\,\text{CSF}}{B_{ijk}}\,\underset{\uparrow\,\text{hash map}}{c_k}$$

```
for (int pA = 0; pA < m*n; pA++) {
  A[pA] = 0.0;
}
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_crd[pB2];
    int pA2 = i*n + j;
    double t = 0.0;
    for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
      int k = B3_crd[pB3];
      int pc1 = k % c_size;
      if (c_crd[pc1] != k && c_crd[pc1] != -1) {
        int end = pc;
        do {
          pc = (pc+1) % c_size;
        } while (c_crd[pc1] != k &&
                 c_crd[pc1] != -1 && pc1 != end);
      }
      if (c_crd[pc1] == k) {
        t += B[pB3] * c[pc1];
      }
    }
    A[pA2] = t;
  }
}
```

# Optimized code is often complex, especially when it iterates over irregular data structures

$$A_{ijk} = \underset{\text{CSF}}{B_{ijk}} + \underset{\text{COO}}{C_{ijk}}$$

```
int iB = 0;
int C0_pos = C0_pos[0];
while (C0_pos < C0_pos[1]) {
  int iC = C0_crd[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos[1]) && (C0_crd[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_crd[B1_pos];
      int jC = C1_crd[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_crd[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_crd[B2_pos];
          int kC = C2_crd[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A[A2_pos] = B[B2_pos] + C[C2_pos];
          } else if (kB == k) {
            A[A2_pos] = B[B2_pos];
          } else {
            A[A2_pos] = C[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos[B1_pos + 1]) {
          int kB0 = B2_crd[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A[A2_pos0] = B[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_crd[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A[A2_pos1] = C[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos[B1_pos];
                 B2_pos0 < B2_pos[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_crd[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A[A2_pos2] = B[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_crd[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A[A2_pos3] = C[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos[iB + 1]) {
      int jB0 = B1_crd[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos[B1_pos];
               B2_pos1 < B2_pos[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_crd[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A[A2_pos4] = B[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_crd[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_crd[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_crd[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A[A2_pos5] = C[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos[iB];
             B1_pos0 < B1_pos[iB + 1]; B1_pos0++) {
      int jB1 = B1_crd[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos[B1_pos0];
               B2_pos2 < B2_pos[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_crd[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A[A2_pos6] = B[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos[iB];
           B1_pos1 < B1_pos[iB + 1]; B1_pos1++) {
    int jB2 = B1_crd[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos[B1_pos1];
             B2_pos3 < B2_pos[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_crd[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A[A2_pos7] = B[B2_pos3];
    }
  }
  iB++;
}
```

Can we get abstractions *without* friction by moving the abstractions into the compiler?

Can we get abstractions *without* friction by moving the abstractions into the compiler?

Domain-Specific
Language
Constructs

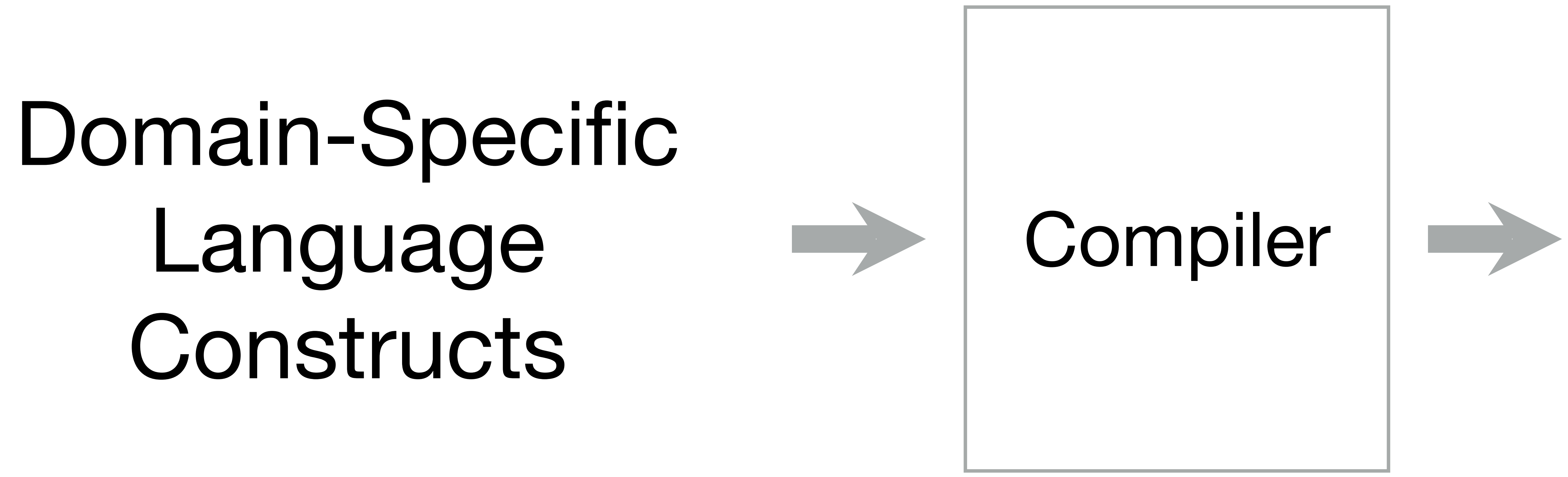
# Can we get abstractions *without* friction by moving the abstractions into the compiler?

Domain-Specific Language Constructs

## Generated Code

```
int iB = 0;
int C0_pos = C0_pos_arr[0];
while (C0_pos < C0_pos_arr[1]) {
  int iC = C0_idx_arr[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos_arr[1]) && (C0_idx_arr[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos_arr[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos_arr[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_idx_arr[B1_pos];
      int jC = C1_idx_arr[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_idx_arr[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos_arr[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos_arr[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_idx_arr[B2_pos];
          int kC = C2_idx_arr[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos] + C_val_arr[C2_pos];
          } else if (kB == k) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos];
          } else {
            A_val_arr[A2_pos] = C_val_arr[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos_arr[B1_pos + 1]) {
          int kB0 = B2_idx_arr[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A_val_arr[A2_pos0] = B_val_arr[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_idx_arr[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A_val_arr[A2_pos1] = C_val_arr[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos_arr[B1_pos];
                 B2_pos0 < B2_pos_arr[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_idx_arr[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A_val_arr[A2_pos2] = B_val_arr[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_idx_arr[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A_val_arr[A2_pos3] = C_val_arr[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos_arr[iB + 1]) {
      int jB0 = B1_idx_arr[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos_arr[B1_pos];
               B2_pos1 < B2_pos_arr[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_idx_arr[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A_val_arr[A2_pos4] = B_val_arr[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_idx_arr[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_idx_arr[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_idx_arr[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A_val_arr[A2_pos5] = C_val_arr[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos_arr[iB];
             B1_pos0 < B1_pos_arr[iB + 1]; B1_pos0++) {
      int jB1 = B1_idx_arr[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos_arr[B1_pos0];
               B2_pos2 < B2_pos_arr[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_idx_arr[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A_val_arr[A2_pos6] = B_val_arr[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos_arr[iB];
           B1_pos1 < B1_pos_arr[iB + 1]; B1_pos1++) {
    int jB2 = B1_idx_arr[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos_arr[B1_pos1];
             B2_pos3 < B2_pos_arr[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_idx_arr[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A_val_arr[A2_pos7] = B_val_arr[B2_pos3];
    }
  }
  iB++;
}
```

# Can we get abstractions *without* friction by moving the abstractions into the compiler?




## Generated Code

```
int iB = 0;
int C0_pos = C0_pos_arr[0];
while (C0_pos < C0_pos_arr[1]) {
  int iC = C0_idx_arr[C0_pos];
  int C0_end = C0_pos + 1;
  if (iC == iB)
    while ((C0_end < C0_pos_arr[1]) && (C0_idx_arr[C0_end] == iB)) {
      C0_end++;
    }
  if (iC == iB) {
    int B1_pos = B1_pos_arr[iB];
    int C1_pos = C0_pos;
    while ((B1_pos < B1_pos_arr[iB + 1]) && (C1_pos < C0_end)) {
      int jB = B1_idx_arr[B1_pos];
      int jC = C1_idx_arr[C1_pos];
      int j = min(jB, jC);
      int A1_pos = (iB * A1_size) + j;
      int C1_end = C1_pos + 1;
      if (jC == j)
        while ((C1_end < C0_end) && (C1_idx_arr[C1_end] == j)) {
          C1_end++;
        }
      if ((jB == j) && (jC == j)) {
        int B2_pos = B2_pos_arr[B1_pos];
        int C2_pos = C1_pos;
        while ((B2_pos < B2_pos_arr[B1_pos + 1]) && (C2_pos < C1_end)) {
          int kB = B2_idx_arr[B2_pos];
          int kC = C2_idx_arr[C2_pos];
          int k = min(kB, kC);
          int A2_pos = (A1_pos * A2_size) + k;
          if ((kB == k) && (kC == k)) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos] + C_val_arr[C2_pos];
          } else if (kB == k) {
            A_val_arr[A2_pos] = B_val_arr[B2_pos];
          } else {
            A_val_arr[A2_pos] = C_val_arr[C2_pos];
          }
          if (kB == k) B2_pos++;
          if (kC == k) C2_pos++;
        }
        while (B2_pos < B2_pos_arr[B1_pos + 1]) {
          int kB0 = B2_idx_arr[B2_pos];
          int A2_pos0 = (A1_pos * A2_size) + kB0;
          A_val_arr[A2_pos0] = B_val_arr[B2_pos];
          B2_pos++;
        }
        while (C2_pos < C1_end) {
          int kC0 = C2_idx_arr[C2_pos];
          int A2_pos1 = (A1_pos * A2_size) + kC0;
          A_val_arr[A2_pos1] = C_val_arr[C2_pos];
          C2_pos++;
        }
      } else if (jB == j) {
        for (int B2_pos0 = B2_pos_arr[B1_pos];
                 B2_pos0 < B2_pos_arr[B1_pos + 1]; B2_pos0++) {
          int kB1 = B2_idx_arr[B2_pos0];
          int A2_pos2 = (A1_pos * A2_size) + kB1;
          A_val_arr[A2_pos2] = B_val_arr[B2_pos0];
        }
      } else {
        for (int C2_pos0 = C1_pos; C2_pos0 < C1_end; C2_pos0++) {
          int kC1 = C2_idx_arr[C2_pos0];
          int A2_pos3 = (A1_pos * A2_size) + kC1;
          A_val_arr[A2_pos3] = C_val_arr[C2_pos0];
        }
      }
      if (jB == j) B1_pos++;
      if (jC == j) C1_pos = C1_end;
    }
    while (B1_pos < B1_pos_arr[iB + 1]) {
      int jB0 = B1_idx_arr[B1_pos];
      int A1_pos0 = (iB * A1_size) + jB0;
      for (int B2_pos1 = B2_pos_arr[B1_pos];
               B2_pos1 < B2_pos_arr[B1_pos + 1]; B2_pos1++) {
        int kB2 = B2_idx_arr[B2_pos1];
        int A2_pos4 = (A1_pos0 * A2_size) + kB2;
        A_val_arr[A2_pos4] = B_val_arr[B2_pos1];
      }
      B1_pos++;
    }
    while (C1_pos < C0_end) {
      int jC0 = C1_idx_arr[C1_pos];
      int A1_pos1 = (iB * A1_size) + jC0;
      int C1_end0 = C1_pos + 1;
      while ((C1_end0 < C0_end) && (C1_idx_arr[C1_end0] == jC0)) {
        C1_end0++;
      }
      for (int C2_pos1 = C1_pos; C2_pos1 < C1_end0; C2_pos1++) {
        int kC2 = C2_idx_arr[C2_pos1];
        int A2_pos5 = (A1_pos1 * A2_size) + kC2;
        A_val_arr[A2_pos5] = C_val_arr[C2_pos1];
      }
      C1_pos = C1_end0;
    }
  } else {
    for (int B1_pos0 = B1_pos_arr[iB];
             B1_pos0 < B1_pos_arr[iB + 1]; B1_pos0++) {
      int jB1 = B1_idx_arr[B1_pos0];
      int A1_pos2 = (iB * A1_size) + jB1;
      for (int B2_pos2 = B2_pos_arr[B1_pos0];
               B2_pos2 < B2_pos_arr[B1_pos0 + 1]; B2_pos2++) {
        int kB3 = B2_idx_arr[B2_pos2];
        int A2_pos6 = (A1_pos2 * A2_size) + kB3;
        A_val_arr[A2_pos6] = B_val_arr[B2_pos2];
      }
    }
  }
  if (iC == iB) C0_pos = C0_end;
  iB++;
}
while (iB < B0_size) {
  for (int B1_pos1 = B1_pos_arr[iB];
           B1_pos1 < B1_pos_arr[iB + 1]; B1_pos1++) {
    int jB2 = B1_idx_arr[B1_pos1];
    int A1_pos3 = (iB * A1_size) + jB2;
    for (int B2_pos3 = B2_pos_arr[B1_pos1];
             B2_pos3 < B2_pos_arr[B1_pos1 + 1]; B2_pos3++) {
      int kB4 = B2_idx_arr[B2_pos3];
      int A2_pos7 = (A1_pos3 * A2_size) + kB4;
      A_val_arr[A2_pos7] = B_val_arr[B2_pos3];
    }
  }
  iB++;
}
```

# Separation of Algorithm, Data Representation, and Schedule

Algorithm
Language

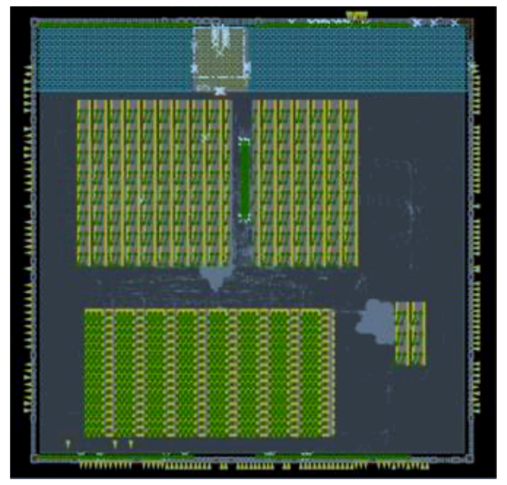
Data Representation
Language

Scheduling
Language

Compiler


CPUs


GPUs



DSAs

Most of HW retargeting is about changing schedules and data representations

# For Discussion

```
c := 0
for i := 1 step 1 until n do
    c := c + a[i]×b[i]
```

```
c := 0
for i := 1 step 1 until n do
   c := c + a[i]×b[i]
```

**Def** Innerproduct
$$\equiv (\text{Insert } +) \circ (\text{ApplyToAll } \times) \circ \text{Transpose}$$

```
c := 0
for i := 1 step 1 until n do
   c := c + a[i]×b[i]
```

**Def** Innerproduct

$$\equiv (\text{Insert } +)\circ(\text{ApplyToAll } \times)\circ\text{Transpose}$$

IP:<<1,2,3>, <6,5,4>> =

| | |
|---|---|
| Definition of IP | ⇒ (/+)∘(α×)∘Trans: <<1,2,3>, <6,5,4>> |
| Effect of composition, ∘ | ⇒ (/+):((α×):(Trans: <<1,2,3>, <6,5,4>>)) |
| Applying Transpose | ⇒ (/+):((α×): <<1,6>, <2,5>, <3,4>>) |
| Effect of ApplyToAll, α | ⇒ (/+): <×: <1,6>, ×: <2,5>, ×: <3,4>> |
| Applying × | ⇒ (/+): <6,10,12> |
| Effect of Insert, / | ⇒ +: <6, +: <10,12>> |
| Applying + | ⇒ +: <6,22> |
| Applying + again | ⇒ 28 |